UNIVERSITÀ DEGLI STUDI DELL'AQUILA

Department of Information Engineering, Computer Science and Mathematics

Ph.D. Program in Information and Communication Technology

Curriculum Emerging computing models, software architectures, and intelligent systems

XXXV ciclo

Conceptualization and Development of ML-based Recommender Systems for Software Engineering

SSD INF/01

Candidate

Claudio Di Sipio

Doctoral Program Supervisor

Prof. Vittorio Cortellessa

Advisor

Prof. Davide Di Ruscio

Co-Advisors

Dott. Juri Di Rocco

Dott. Phuong T. Nguyen

a.y. 2021/2022

Acknowledgements

Everything must come to an end, even though it does not mean that we cannot start over by pursuing our passion. This long journey culminates in this dissertation, and it is time to thank all the people that contribute to making it possible.

First of all, I would like to express my gratitude to my advisor, Prof. Davide Di Ruscio, for his invaluable support (and patience) during these years. Since my master's degree, he inspired me to follow the academic path by being a real mentor rather than just my supervisor.

Second, I thank my co-advisor Dott. Juri Di Rocco for helping me, especially with the technical part of this huge work. We worked side by side many times, and he was always available to solve any doubts.

I would like to thank my co-advisor Dott. Phuong Thanh Nguyen for his hard work in revising this dissertation and his insightful writing tips.

A heartfelt thank goes to Riccardo Rubei, a friend more than just a colleague. We went through these years by sharing passions and having engaging discussions beyond the academic world. Altogether, I really felt part of this great team that works cohesively to achieve remarkable goals.

A very special thank goes to my friends and colleagues in Coppito, i.e., Roberta, Gianluca, Luca, Andrea B., and Walter. Despite the pandemic, we enjoyed these years by sharing our ideas, doubts, and knowledge. Although being a Ph.D. student might be hard sometimes, their sight helps me a lot during these years. An *ad personam* consideration is due to Giordano who has been also my housemate here in L'Aquila. We knew each other for several years and I wish him all the best for the academic path that he started recently. A shoutout goes also to the new Ph.D. students in Coppito, i.e., Mashal, Gennaro, Federico, and Andrea D'A. Even though we met for a short time, I felt their passion and I wish them all the bests.

Above all, I would thank Isabella just for being the person she is. Despite she shares the last part of this journey, she was by my side, supporting me when needed.

An additional thank goes to my friends in Chieti, including Francesca, Silvia, Valentina, Stefano and Simone. Despite the distance, we managed to keep in touch during these years.

Last but not least, I feel I owed a debt of gratitude to my father Pantaleone, my mother Daniela, and my aunt Antonella. They supported me in every decision, pushing me to follow my desires no matter what ever happens.

Abstract

To perform their daily tasks, developers intensively make use of existing resources by consulting open-source software (OSS) repositories. Such platforms contain rich data sources, e.g., code snippets, documentation, and user discussions, that can be useful for supporting development activities. Over the last decades, several techniques and tools have been promoted to provide developers with innovative features, aiming to bring in improvements in terms of development effort, cost savings, and productivity. Recommender systems (RSs) are complex software systems that suggest relevant items of interest given a specific application domain to users. The development of RSs encompasses the execution of different steps, including data preprocessing, choice of appropriate algorithms, and item delivery, to name a few. Though RSs can alleviate the curse of information overload, existing approaches resemble black-box systems, where the end-user is not supposed to customize the overall process.

This dissertation aims to advance the current state-of-the-art by conceptualizing a series of recommender systems to support software developers. Furthermore, we also investigate different types of adversarial attacks that these systems may encounter. Finally, we propose an MDE-based tool that automatizes the design, fine-tuning, and deployment of any recommender system.

By relying on the experience gained in the CROSSMINER project, we elicit foundational aspects of RSs that come in handy in defying essential components of a generic RS. We investigate the feasibility of cutting-edge technologies applied to the RS domain, i.e., machine learning, stochastic networks, and natural process languages (NLP) strategies, by proposing recommendation systems to support developers in different SE tasks, including API function calls, categorization of GitHub projects, and modeling activities.

Afterward, a tailored metamodel has been built to generate the actual system using MDE-based technology, following the low-code paradigm to democratize the usage of the RS. The proposed tool called LEV4REC is capable of resembling existing systems in terms of different metrics.

Altogether, we elicit the fundamental components relying on the prior knowledge matured in developing actual recommender systems. Such experience has been used to develop an

MDE-based tool specifically conceived to reduce the overall effort for newcomers users who do not have prior knowledge of such systems. Although there is still room for improvement, all the proposed approaches in this dissertation succeeded in providing decent recommendations for the covered application domains, thus facilitating the completion of various software engineering tasks.

Table of contents

List of figures **xi**

List of tables **xv**

1 Introduction **1**

1.1 Identified challenges . . . 2
1.2 Research objectives . . . 3
1.3 Structure of the dissertation . . . 6

2 Literature review **7**

2.1 Existing recommender systems for SE . . . 7
2.1.1 Filtering techniques to recommend API . . . 7
2.1.2 Automatic approaches to classify OSS projects . . . 9
2.1.3 Modeling assistant tools . . . 12
2.2 Democratizing the development of complex systems . . . 15
2.2.1 Model-driven engineering and low-code development platforms . . . 15
2.2.2 Supporting the automatic design of recommender system . . . 16
2.2.3 Automatic machine learning . . . 18

3 A conceptual framework to develop RSSEs **21**

3.1 RSSEs bird-eye architecture . . . 22
3.2 Main RSSE design features . . . 25
3.3 Evaluation metrics for RSSE . . . 30
3.3.1 Terminology . . . 30
3.3.2 Metrics . . . 31
3.4 Conclusion . . . 34

4 Recommending API function calls to support developers **35**

4.1 FOCUS . . . 37

4.1.1 Motivation and background 37
4.1.2 FOCUS architecture 39
4.1.3 Evaluation 47
4.1.4 Results 59
4.1.5 Threats to validity 69
4.2 LUPE 70
4.2.1 Machine translation with Encoder-Decoder 70
4.2.2 LUPE architecture 71
4.2.3 Evaluation 77
4.2.4 Results 81
4.2.5 Threats to validity 91
4.3 Conclusion 92

5 Categorizing GitHub projects **93**
5.1 MNBN 95
5.1.1 Motivation and background 95
5.1.2 The MNBN approach 98
5.1.3 Evaluation 101
5.1.4 Results 103
5.1.5 Threats to validity 108
5.2 HybridRec 109
5.2.1 HybridRec architecture 109
5.2.2 Evaluation 116
5.2.3 Evaluation process 117
5.2.4 Results 119
5.2.5 Threats to validity 131
5.3 Conclusion 131

6 Assisting modelers in specifying models and metamodels **133**
6.1 MemoRec 135
6.1.1 Motivation and background 135
6.1.2 MemoRec architecture 138
6.1.3 Evaluation 146
6.1.4 Results 149
6.1.5 Threats to validity 157
6.2 MORGAN 158
6.2.1 Graph kernel similarity 158

6.2.2 MORGAN architecture 159
6.2.3 Evaluation 166
6.2.4 Experimental results 173
6.2.5 Threats to validity 186
6.3 Conclusion 187

7 Challenges and lessons learned from the conceived RSSEs 189
7.1 Challenges and lessons learned related to requirements elicitation 190
7.1.1 Lessons learned 193
7.2 Challenges and lessons learned related to development 195
7.2.1 Lessons learned 201
7.3 Challenges and lessons learned related to evaluation 203
7.3.1 Lessons learned 210
7.4 Conclusion 212

8 An MDE-based methodology to engineering recommender systems 215
8.1 Supporting technologies 216
8.2 The LEV4REC environment 217
8.2.1 RS Feature Selection 218
8.2.2 RS Feature Configuration 220
8.2.3 RS Code Generation 224
8.2.4 Deploying LEV4REC 225
8.3 Use cases 230
8.4 Evaluation strategies 235
8.4.1 Research questions 236
8.4.2 Datasets 236
8.4.3 Metrics 237
8.4.4 Automated evaluation 237
8.4.5 Focus group evaluation 238
8.5 Results 240
8.6 Threats to validity 247
8.7 Conclusion 248

9 Adversarial Attacks to Recommender Systems in Software Engineering 249
9.1 Adversarial attacks to TPL RSSEs 252
9.1.1 Motivation and background 252
9.1.2 Proof of concept 254

9.1.3 Experimental results 257
9.2 Adversarial attacks to API RSSEs 259
9.2.1 Push attacks to API and code snippet RSSE 259
9.2.2 Study design and planning 263
9.2.3 Results 267
9.2.4 Threats to validity 274
9.2.5 Discussions 275
9.3 Conclusion 278

10 Conclusion 279
10.1 Summary of the contributions 279
10.2 Publications 282
10.3 Future work 287

References 291

List of figures

1.1 Dissertation conceptual map. 4

3.1 Bird-eye architecture of developed RSSEs. 23
3.2 Main design features of recommendation systems in software engineering. . 27

4.1 The FOCUS motivating example. 37
4.2 Overview of the FOCUS architecture. 40
4.3 Rating matrix for a project with 4 declarations and 4 invocations. 42
4.4 FOCUS IDE. 45
4.5 The data extraction process. 48
4.6 The extraction of data for a testing project. 50
4.7 Precision and recall curves obtained for the configurations. 63
4.8 Bivariate analysis of Precision and Cardinality. 64
4.9 Levenshtein distance for the set of 500 apps. 67
4.10 Results for evaluating the usefulness of the recommendations. 68
4.11 LSTM and Encoder-Decoder. 70
4.12 System architecture. 72
4.13 Method declaration–Method invocation pair. 73
4.14 Input and output API sequences for the example in Figure 4.1b and Table 4.7. 74
4.15 Training with reversed input sequence X=“*4␣21␣8␣5*” and output sequence Y=“*4␣15␣23␣27␣55␣81␣41*”. 75
4.16 Data encoding with matrix and tensor. 75
4.17 LUPE Precision with configuration C_1. 82
4.18 LUPE Recall with configuration C_1. 83
4.19 Combination of the scores from all the folds with configuration C_1. 83
4.20 Combination of the scores from all the folds with configuration C_2. 83
4.21 LUPE Precision with configuration C_3. 84
4.22 LUPE Recall with configuration C_3. 84

4.23 Comparison between GAPI and LUPE. 86
4.24 LUPE Precision recall on $\mathbf{D}_3$. 89
4.25 LUPE Precision recall on $\mathbf{D}_{3S}$. 89

5.1 The *longclaw* repository from GitHub with different topics. 96
5.2 Overview of the proposed approach. 98
5.3 Distribution of the number of topics in the dataset. 103
5.4 The testing phase for a single project. 104
5.5 Success rate for c=2. 106
5.6 Success rate for c=5. 107
5.7 Success rate for c=8. 107
5.8 Overview of the HybridRec approach. 109
5.9 The **ST** component. 110
5.10 The **CF** component. 113
5.11 Graph representation for repositories and topics. 115
5.12 Evaluation process. 118
5.13 Success rate for D_1 considering $N = \{1,\dots,10\}$. 122
5.14 Success rate values for $N = \{1,5,10,15,20\}$. 123
5.15 Precision/recall curves. 124
5.16 Success rate on MVN Repository dataset. 126
5.17 Precision recall curve on the MVN Repository dataset. 127
5.18 Catalog coverage on the MVN Repository dataset. 127
5.19 The distribution of topics in the D_1 dataset. 129

6.1 The illustrative UMLDSL metamodel. 136
6.2 The explanatory SPRINGBOOT model. 136
6.3 The explanatory APACHE STREEMS PROVIDER JSON Schema. 137
6.4 Overview of MemoRec's architecture. 139
6.5 The Web metamodel data extraction. 141
6.6 Matrix representation of metamodels w.r.t. structural features and classes. . 143
6.7 Graph representation of metamodels and structural features. 144
6.8 Evaluation Process. 149
6.9 Dataset $\mathbf{D}_1$ 153
6.10 Dataset $\mathbf{D}_2$ 153
6.11 Average execution time. 153
6.12 The `Bibtex` metamodel. 154
6.13 The MORGAN architecture. 160

6.14 Example of a stemmed graph. 163
6.15 D_α dataset creation and overall evaluation process. 165
6.16 D_α features. 168
6.17 D_β features. 168
6.18 D_γ features. 169
6.19 D_δ features. 169
6.20 D_ε features. 170
6.21 Evaluation scores for recommending model classes. 175
6.22 Evaluation scores for recommending class members. 176
6.23 Evaluation scores for recommending metaclasses. 177
6.24 Evaluation scores for structural features. 178
6.25 Evaluation scores for recommending JSON root properties. 184
6.26 Evaluation scores for recommending JSON nested properties. 185

7.1 Map of challenges and lessons learned. 190
7.2 `pom.xml` files of a project before (*left*) and after (*right*) having adopted third-party libraries recommended by CrossRec. 191
7.3 Recommended API calls for the `getScoresFromLivescore()` method in Listing 7.1. 193
7.4 Requirement definition process. 194

8.1 Overview of the proposed approach. 218
8.2 Overview of the proposed feature model. 219
8.3 The LEV4REC configuration metamodel. 221
8.4 The LEV4REC `Dataset` and `DataStructure` metaclasses. 222
8.5 The LEV4REC `RecommenderSystem` metaclass. 223
8.6 The LEV4REC `ValidationTechnique` metaclass. 223
8.7 The LEV4REC `PresentationLayer` metaclass. 225
8.8 Using LEV4REC RSs from IDEs. 226
8.9 The LEV4REC web-based editor architecture. 227
8.10 A fragment of the RS configuration form. 228
8.11 The DSL web editor. 229
8.12 Small fragment of the KNN specification. 231
8.13 Small fragment of the AURORA specification. 232
8.14 The evaluation process. 238
8.15 Precision of the KNN-based RS developed with LEV4REC. 241
8.16 Recall of the KNN-based RS developed with LEV4REC. 241

8.17 F_1 score of the KNN-based RS developed with LEV4REC. 242
8.18 Precision of AURORA as developed with LEV4REC. 243
8.19 Recall of AURORA as developed with LEV4REC. 243
8.20 F_1 score of AURORA as developed with LEV4REC. 244

9.1 Manipulation of library usages for a fake project. 255
9.2 The process of manipulating GitHub to promote malicious repositories. . . 263
9.3 Frequency of APIs in projects and declarations. 267
9.4 Number of papers for the related topics. 269

List of tables

3.1 The examined use cases. 24

4.1 Experimental configurations. 52
4.2 Task assignments to the evaluator groups. 56
4.3 Success rate of PAM and FOCUS. 59
4.4 Success rate (%) for $k = \{2,3,4,6,10\}$ and $N = \{1,5,10,15,20\}$. 60
4.5 Performance gain (%) among the configurations. 61
4.6 Correlation (ρ) between cardinality and precision, $N = \{5,10,15,20,25\}$. . 65
4.7 Input, outputs APIs and their corresponding IDs. 74
4.8 Datasets. 78
4.9 Experimentation settings. 80
4.10 LUPE Success rate. 81
4.11 Percentage of definitions getting 0 as Levenshtein distance (%). 84
4.12 Results obtained by FACER on DS_2. 87
4.13 $\mathbf{RQ}_3$: Success rate and percentage of recommendations getting 0 as Levenshtein distance (ρ_L) on $\mathbf{DS}_3$. 88
4.14 $\mathbf{RQ}_3$: Success rate and percentage of recommendations getting 0 as Levenshtein distance (ρ_L) on $\mathbf{DS}_3$. 88

5.1 The *longclaw* repository topics. 96
5.2 Example of repositories, their topics and the recommended topics. 105
5.3 Statistics 105
5.4 Precision, recall, and top-rank. 108
5.5 The *artifact-topic matrix* for the example. 114
5.6 Datasets features. 118
5.7 Summary 120
5.8 Success rate, precision, and recall. 120
5.9 Precision, recall with D_1. 122

5.10 Catalog coverage. 125
5.11 Metadata used by the **ST** and **CF** components. 129
5.12 Configuration parameters of the **ST** and **CF** components. 130

6.1 *Package-class* feature rating matrix combined with SE_c for the Web metamodel. 142
6.2 *Class-structural feature* rating matrix combined with IE_s for the Web metamodel. 142
6.3 ϕ vectors for the metamodels depicted in Fig. 6.7. 145
6.4 sim_1 matrix for the metamodels depicted in Fig. 6.7. 145
6.5 Datasets. 148
6.6 Success rate for structural feature recommendations, $k = \{1, 5, 10, 15, 20\}$, by considering the $\mathbf{D}_1$ dataset. 150
6.7 Success rate for class recommendations, $k = \{1, 5, 10, 15, 20\}$, by considering the $\mathbf{D}_1$ dataset. 151
6.8 Success rate, $k = \{1, 5, 10, 15, 20\}$, by comparing the adoption of $\mathbf{D}_1$ and $\mathbf{D}_2$ 152
6.9 Precision values 152
6.10 Recall values 153
6.11 F-measure values 154
6.12 Predicted recommendations for the `Article` metaclass and `Bibtex` package of the metamodel in Fig. 6.12. 155
6.13 Success rate, precision and recall for class recommendations 157
6.14 Retrieved items for the UMLDSL metamodel. 165
6.15 Configuration settings. 172
6.16 Average prediction scores, Wilcoxon rank sum test adjusted *p*-values and Cliff's *d* results. 180
6.17 Timing performance on D_α (seconds). 181
6.18 Timing performance on D_β (seconds). 181
6.19 Comparison between C_1 and C_3 considering D_α. 182
6.20 Comparison between C_1 and C_3 considering D_β. 182
6.21 Evaluation scores for D_δ. 185
6.22 Evaluation scores for D_ε. 186

7.1 RSSEs evaluation facts. 208

8.1 Statistics 220
8.2 Overview of the examined datasets. 237
8.3 Participants to the focus group and their expertise. 239
8.4 Comparison with original results. 242

8.5 Identified themes and sub-themes using for the TAT methodology. 245

9.1 Notable RSSE for mining libraries and APIs. 253
9.2 Hit ratio @N for LibRec. 256
9.3 Hit ratio @N for CrossRec. 257
9.4 Notable RSSE for mining APIs and/or code snippets (Listed in chronological order). 268
9.5 Keywords. 269
9.6 Hit ratio for the recommendations by UP-Miner. 273
9.7 Hit ratio for the recommendations by PAM. 273
9.8 Hit ratio for the recommendations by FOCUS. 273

Chapter 1

Introduction

Over the last decades, software developers have made use of reusable components to ease the burden of reimplementing complex functionalities from scratch. Despite the availability of a huge amount of data, users have experienced the so-called over-choice problem, thus getting frustrated by the impressive solution space that is still growing. In this respect, the proliferation of cutting-edge technologies can reduce the burden of choice by automatizing the required activities, i.e., requirements gathering, system design, algorithm selection, and evaluation of the system. In particular, recommender systems in software engineering (RSSEs) are at the forefront of supporting users in this decision task by retrieving lists of suitable items starting from an initial context [89, 213, 220]. Though being useful, these systems necessitate deep knowledge of different technologies as well as extra-development activities bounded by specific programming languages. Moreover, developers are not allowed to personalize them to address specific SE tasks, possibly uncovered by the existing techniques. Therefore, frameworks that guide developers in specifying the whole recommender pipeline from scratch are gaining momentum, pushing toward the automation of the entire design flow that includes the actual development and the assessment of the produced system [13, 19, 123, 280]. To this end, the majority of them exploits machine-learning (ML), artificial intelligence (AI), and low-code development platforms (LCDP) to simplify the development of complex software systems. On the one hand, such frameworks offer a set of utilities to allow for the customization of every aspect of the process through user-friendly configuration interfaces. On the other hand, the support for the constraints specification at design time is not fully covered. More importantly, while effort has been made to make recommender systems more accurate, the issue of protecting them from adversarial attacks has been greatly neglected [187, 188].

This dissertation is threefold. *First*, we conceived a series of RSSEs to assist developers in their daily tasks, providing them with various artifacts such as code snippets or metamodel

components. Moreover, we present a generic framework to conceptualize generic RSSEs by relying on the experience gained under the umbrella of the CROSSMINER [67] concerning the challenges and lessons learned from the development of these systems. The experiences gained from the developed RSSEs allow us to elicit components and process that are common in the recommender system development process. *Second*, the conducted conceptualization eventually culminates in the development of an MDE-based tool, named LEV4REC [73], aiming to facilitate the design and the deployment of such systems. *Third*, we present an initial investigation on possible adversarial attempt to harm RSSEs [187, 188], triggering the need for proper countermeasures.

1.1 Identified challenges

Even though several solutions are in place, supporting the development of RSs poses a set of challenges and issues that are not covered yet. In the scope of this dissertation, we identified the following challenges:

- *CH1: Analyzing and parsing complex software project.* The knowledge needed to manipulate an API can be extracted from various sources: the API source code itself, the official website and documentation, Q&A websites such as Stack Overflow, forums and mailing lists, bug trackers, other projects using the same API, etc. However, official documentation often merely reports the API description without providing non-trivial example usages. Besides, querying informal sources such as StackOverflow might become time-consuming and error-prone [218]. Also, API documentation may be ambiguous, incomplete, or erroneous [263], while API examples found on Q&A websites may be of poor quality [173].

- *CH2: Mining and preprocessing textual data from GitHub.* Over the last decade, open-source software repositories have gained a prominent role in storing and managing software projects. Different kinds of artifacts, including source code, mailing lists, bug tracking systems, and documentation, are stored and managed homogeneously employing powerful technologies. In such a context, GitHub is playing the role of forefront platform managing more than 190M repositories, with more than 28M of them being available to the public. Even though there are dedicated APIs, extracting the repositories' data is still challenging due to several issues, e.g., limitations in the number of the requests, heterogeneous source of information, to name a few.

- *CH3: Encoding model and metamodels features.* During their daily tasks, modelers might expect to get recommendations consisting of relevant metaclasses or structural

features that can be further integrated. However, due to a huge amount of available resources, searching for suitable artifacts is a daunting task. Under the circumstances, we see an urgent need for suitable machinery to mine and encode data from open source platforms such as GitHub. Among others, we are interested in finding which packages and metaclasses can be added to the metamodel under development, given that other packages and metaclasses are already defined.

- *CH4: Understanding how recommender systems can be deceived.* While these systems have proven to be effective in terms of prediction accuracy, there has been less attention for what concerns such recommenders' resilience against adversarial attempts. In fact, by crafting the recommenders' learning material, e.g., data from large open-source software (OSS) repositories, hostile users may succeed in injecting malicious data, putting at risk the software clients adopting API recommender systems.

- *CH5: Supporting the design and deployment of generic recommender systems.* While the issue of designing and implementing generic recommender systems has been carefully addressed by state-of-the-art studies [207, 220], there is a lack of proper references for the design of *generic* recommendation systems. Being built on top of various components, including data preprocessing, choice of suitable algorithms, item delivery, RSs can ease the curse of information overload, enabling developers to approach the most relevant artifacts while they are coding. Nevertheless, existing approaches resemble black-box systems, where the end-user is not allowed to customize the overall process. In this respect, there is the urgent need for proper tools to help developers design, build and evaluate their own RSs.

1.2 Research objectives

The conceptual map depicted in Figure 1.1 summarizes the content of this dissertation in terms of challenges and proposed solutions.

RO1: Supporting developers with API function calls. While implementing software projects, developers do not reinvent the wheel but try to reuse existing API calls and source code. In recent years, the problems related to recommending APIs and code snippets have been intensively investigated. Although current approaches have achieved encouraging performance, there is still the need to improve the recommendation process's effectiveness and efficiency. To this end, we exploit a context-aware collaborative filtering technique to retrieve relevant API function calls by encoding the mutual relationships among projects using

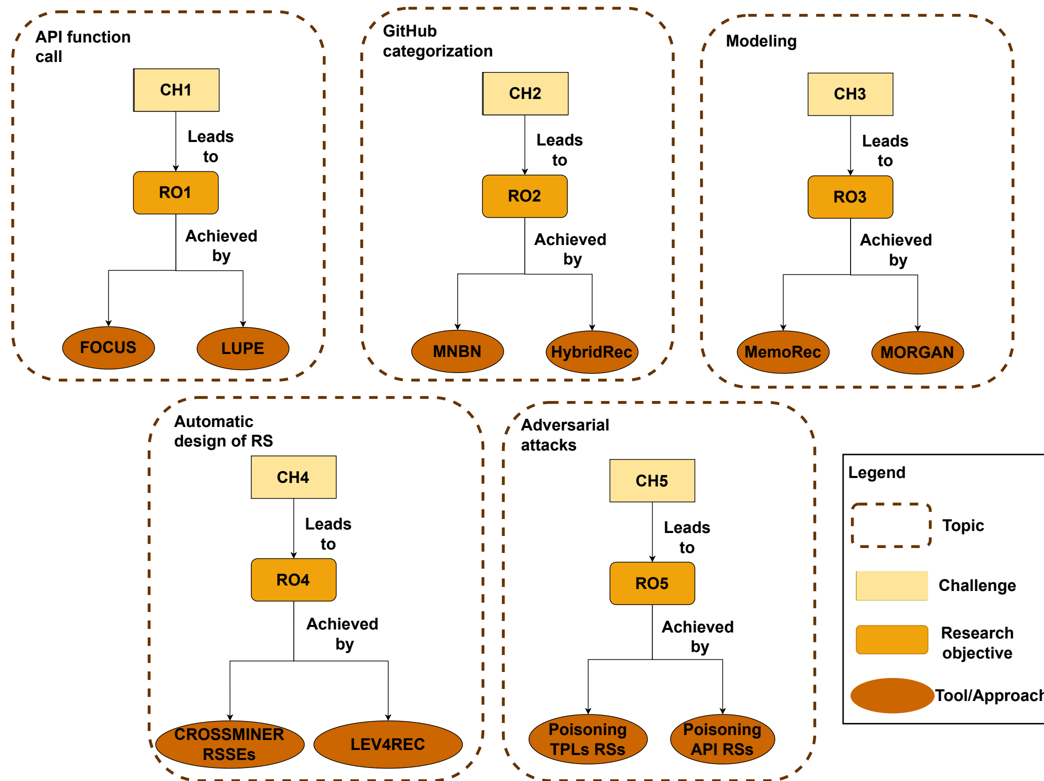


Fig. 1.1 Dissertation conceptual map.

a tensor and mined API usage from the most similar projects. Furthermore, we investigated the usage of sequence-to-sequence neural networks to overcome some well-known issues of deep learning approaches, i.e., setback in timing efficiency [145], and less efficiency when dealing with a long list of APIs.

RO2: Categorizing GitHub projects by exploiting textual content. The platform offers developers the possibility to classify the stored artifacts by means of topics, i.e., it introduced the possibility of labeling the stored repositories only in 2017 to help developers increase the reachability of their repositories. The assigned labels allow users to characterize projects, e.g., with respect to the provided functionalities and employed technologies. However, assigning wrong labels to a given repository can compromise its popularity [37], and even worse, prevent developers from finding projects that they might be willing to contribute. We propose a multi-label classifier based on Multinomial Naive Bayesian network that is capable of recommending featured GitHub topics given the README file. Even though it works in practice, it suffers from some issues, i.e., the approach cannot handle unbalanced data. To overcome such limit, we combine an enhanced version of the network with a collaborative filtering technique, thus increasing the coverage of the suggested items.

RO3: Supporting the specification of models and metamodels. Recently, cutting-edge technologies such as neural networks and NLP techniques have attracted considerable attention from the modeling community [42, 284] as they promote the usage of the so-called intelligent modeling assistants (IMAs) [171]. Altogether, this aims to facilitate the completion of models by providing modelers with insightful artifacts, such as attributes or relationships. Nevertheless, there is still the need to support the automation of modeling activities by offering a convenient way to specify metamodels and models, especially for modelers who are less experienced and need more informative suggestions to complete the tasks at hand. We support the model completion tasks by conceiving two IMAs based on different techniques, i.e., graph kernels and context-aware collaborative filtering algorithm.

RO4: Measuring the impact of adversarial attacks in RSs. Research on Adversarial Machine Learning [114, 262] (AML) studies security issues in Machine Learning systems as well as general-purpose recommender systems [62]. So far, AML has been investigated in different domains, e.g., online systems [164], image classification [174], and addresses both threats and countermeasures [18, 277]. However, to the best of our knowledge, the issue of AML in recommender systems remains unexplored in software engineering. We provide two empirical investigations into the relevance and effects of AML in recommender systems for software engineering (RSSE) by considering two different application domains, i.e., TPL and API recommender system. In the former study, we measure to what extent existing TPLs recommenders are prone to malicious attacks. Similarly, in the second study we experiment with three different API RSSEs by poisoning their training data to measure the impact of the fake APIs in the recommendation list.

RO5: Simplifying the customization and the deployment of RSs using MDE paradigm. While RSs are becoming ubiquitous, we believe that it is necessary to ensure that the next generation of engineers will have a clear understanding of the fundamental techniques and tools underpinning the development. Such engineering process necessitate deep knowledge of different technologies as well as extra-development activities bounded by specific programming languages. To cope with these limitations, frameworks that drive the developer in specifying from scratch the whole recommender pipeline are flourishing, pushing toward the automation of the entire design flow that includes the actual development and the assessment of the produced system [13, 19, 280, 123]. On one hand, they offer a set of utilities that allow the customization of every aspect of the process through user-friendly configuration interfaces. On the other hand, the support for the constraints' specification at design time is not fully covered. First, we elicit common features and components by relying on the

experience gained in the CROSSMINER EU project, in which we developed a set of recommender systems to support several SE tasks. Afterward, we present an automatic approach to assist the developer during the specification, fine-tuning, and deployment of a custom recommender system.

1.3 Structure of the dissertation

Chapter 2 overviews notable recommender systems that support different SE tasks and approaches that simplify the development of complex software systems. In Chapter 3, we present a conceptual framework for developing RSSEs by eliciting notable challenges. Furthermore, we report the gained experience in the frame of the CROSSMINER EU in terms of developed RSSEs. Afterward, we present in Chapter 4 two recommender systems, i.e., FOCUS and LUPE, that retrieve relevant API function calls given the active context. Similarly, Chapter 5 and Chapter 6 show our conceived solutions to categorize GitHub projects and support modeling activities respectively. Chapter 7 discusses challenges and lessons learned from RSSEs. Chapter 8 describes LEV4REC, an MDE-based tool that facilitates the design, the fine-tuning, and the deployment of RSs by relying on specifically-conceived modeling artifacts. We assess both quantitative and qualitative aspects of the tools by conducting two different evaluations, aims at discussing the strengths and limitations of the proposed solution. In Chapter 9, we present two initial investigations in the field of adversarial attacks by considering two types of recommended items, i.e., TPLs and API function calls. We eventually conclude the dissertation in Section 10 by summarizing the contributions of this work and envisioning possible future work in the domain.

Chapter 2

Literature review

2.1 Existing recommender systems for SE

2.1.1 Filtering techniques to recommend API

Acharya *et al.* [10] present a framework to extract API patterns as partial orders from client code. To this aim, control-flow-sensitive static API traces are extracted from source code and sequential patterns are computed. Our approach is able to recommend both a list of API calls and related source code. Zhong *et al.* implemented MAPO, a tool to retrieve API usage patterns from client projects [298]. MAPO extracts API usages from source files, and groups API methods into clusters. Afterwards, it mines API usage patterns from the clusters, ranks them according to their similarity with the current development context, and eventually suggests code snippets to developers. Similarly, UP-Miner [276] extracts API usage patterns by relying on *SeqSim*, a clustering strategy that reduces patterns redundancy as well as improves coverage. While these approaches are based on clustering techniques, and they consider all client projects in the mining regardless of their similarity with the current project, FOCUS narrows down the search scope by looking into similar projects. PAM (Probabilistic API Miner) mines API usage patterns based on a parameter-free probabilistic algorithm [83]. The tool uses the structural Expectation-Maximization (EM) algorithm to infer the most probable API patterns from client code.

NCBUP-miner (Non Client-based Usage Patterns) [228] is a technique that identifies unordered API usage patterns from the API source code, based on both structural (methods that modify the same object) and semantic (methods that have the same vocabulary) relations. The same authors also propose MLUP [227], which is based on vector representation and clustering, but in this case client code is also considered.XSnippet [224] suggests relevant code snippets starting from the developer's context. The system invokes different queries that

consider both the parents of the class and the lexical visible type. Then, queries computed in such a way are passed to a module that mines relevant paths by relying on a graph-based structure. The ranking module eventually ranks the obtained snippets by employing six different heuristics.

DeepAPI [101] is a deep-learning method used to generate API usage sequences given a query in natural language. The learning problem is encoded as a machine translation problem, where queries are considered the source language and API sequences the target language. Only commented methods are considered during the search. The same authors [102] present CODEnn (COde-Description Embedding Neural Network), where, instead of API sequences, code snippets are retrieved to the developer based on semantic aspects such as API sequences, comments, method names, and tokens.

Strathcona [111] is an Eclipse plug-in that extracts the structural context of code and uses it as a query to request API usages from a remote repository. The system performs the match by employing six heuristics associated to class inheritance, method calls, and field types. In a similar fashion, the technique proposed by *Buse and Weimer* [43] synthesizes API examples for a given data type. An algorithm based on data-flow analysis, k-Medoids clustering and pattern abstraction is designed. Its outcome is a set of syntactically correct and well-typed code snippets where example length, exception handling, variables initialization and naming, abstract uses are considered.

MUSE (Method USage Examples) is an approach proposed by *Moreno et al.* [167] to recommending code examples a given API method. MUSE extracts API usages from client code, simplifies code examples with static slicing, and detects clones to group similar snippets. It also ranks examples according to certain properties, i.e., reusability, understandability, and popularity.

SWIM (Synthesizing What I Mean) [208] seeks API structured call sequences (control and data-flows are considered), and then synthesizes API-related code snippets according to a query in natural language. The underlying learning model is also built with the EM algorithm. Similarly, Raychev *et al.* [211] propose a code completion approach based on natural language processing, which receives as input a partial program and outputs a set of API call sequences filling the gaps of the input. Both invocations and invocation arguments are synthesized considering multiple types of an API.

Thummalapenta and Xie propose SpotWeb [257], an approach that provides starting points (hotspots) for understanding a framework, and highlights where examples finding could be more challenging (coldspots). Other tools exploit StackOverflow discussions to suggest code snippets and documentation [54, 202, 204, 209, 217, 254, 261].

GAPI [145] suggests API usage by relying on a graph neural network with the GRU attention mechanism. As the first step, GAPI encodes both high-order API call interactions and software project structural information. Then, GRUs are used to embed the extracted data in the graph-based format. GAPI recommends a list of API usage ranked by their similarity.

Similarly, FACER [9] has been proposed to support Android developers by retrieving API usage for opportunistic reuse. Given a set of open source Java projects, FACER builds a code fact repository that contains the methods' textual body, call graphs, and API usages. Afterward, a clustering technique based on Lucene is employed to retrieve the unique features based on the API similarity. FACER eventually recommends a list of function call usages using frequent pattern mining strategy.

Even though the aforementioned approaches work in practice, there is still room for improvement in terms of offering cohesive API function call. In this respect, this dissertation presents two different API RSSEs i.e., FOCUS and LUPE. In particular, FOCUS exploits a context-aware collaborative filtering technique to provide relevant API code snippets. The conducted evaluation shows that our tool outperforms two notable approaches, i.e., UP-Miner and PAM. We go a step further with LUPE by proposing a sequence-to-sequence neural network to recommend cohesive API function calls. Similar to FOCUS, LUPE has been compared with both GAPI and FACER to showcase its contributions.

2.1.2 Automatic approaches to classify OSS projects

Before the introduction of topics, GRETA (Graph-Based Tag Assignment for GitHub Repositories) was the very first attempt to automatize GitHub tagging [44]. This approach exploited two concepts, namely ETG (entity-tag graph) and a random walk with a restart algorithm. The former is a graph that uses StackOverflow and GitHub data. In contrast, the latter is an algorithm that iteratively explores the global structure of the network to estimate the proximity (affinity score) between two nodes. This aims to build the graph by linking GitHub repositories with StackOverflow posts which share user and lexical similarities. Afterwards, tags are propagated using the walk algorithm.

Repo-Topix [84] was released right after the introduction of GitHub topics as a means to automatically recommend topics by relying on README files and the textual content of a given repository. Standard NLP techniques and a regression model have been applied at the early stage of the process to exclude biased terms from the recommendation process. In addition, the tool makes use of an adapted version of the Jaccard distance to enhance the quality of retrieved items further. Repo-Topix has been preliminarily evaluated using the n-gram ROUGE-1 metrics to count the number of overlapping terms between the recommendation items and the original repository description.

Recently, Izadi *et al.* [120] proposed Repologue, a multi-label classification tool that combines various state-of-the-art classifiers, i.e., MNB, logistic regression, FastText, and DistilBERT. The first step involves topic augmentation using hand-written association rules to derive more information from a given repository and its tags. Then, Repologue compares the mentioned models employing several word embeddings techniques to discover the best one. GHTRec [299] exploits the BERT model to recommend personalized trending repositories[1] by using GitHub topics. First, the underpinning neural network is fed with preprocessed README files to predict topics given a repository. Afterwards, the system captures the user's topic preferences by relying on its commits. The retrieved list is eventually reranked by computing two similarity methods on the topic vectors, i.e., cosine similarity and shared similarity between the developer and a trending repository.

Sally [270] is an automated approach aiming to categorize Maven projects by relying on bytecode analysis and tags extracted from StackOverflow. Given a JAR file as the input, the tool obtains relevant data related to the project, i.e., class names, class fields, and method names. Such results are filtred considering tags excerpted from StackOverflow. In parallel, a weighted graph is computed to identify and encode project dependencies. Then, primary tags are computed by considering only the input project. In contrast, the weighted graph is used to obtain dependencies' tags. These two kinds of tags are combined to retrieve final recommendations in a tag cloud format.

Velázquez-Rodríguez and De Roover [272] have recently proposed MUTAMA, an automated multi-label tagging approach to support Maven projects. Different from our approach, the project's tags are extracted from bytecode, employing a well-founded tool. For each project, the tool takes as the input a pair of groupId-artifactId-version and the corresponding set of tags. In such a way, MUTAMA learns which projects have been tagged similarly by considering the Java classes and methods employed. Then, several machine learning algorithms provided by the MEKA tool are fed with the extracted tags and the vectors obtained from the analyzed source code.

Zhang *et al.* [294] solved the task of keyword-driven hierarchical classification, proposing a tool with three main modules as follows: *(i)* HIN (Heterogeneous Information Network) construction and embedding; *(ii)* key-word enrichment; and *(iii)* topic modeling and pseudo document generation. During the first step, HIN creates a graph to capture all the interactions between the core elements of GitHub repositories, like *Users*, *Words*, *Names*, to name a few. The keyword enrichment step is devised to cope with the problem of scarcity and bias of the keyword provided by users. The machinery in the last step is employed to feed a classifier with pre-labeled repositories and the overfitting related to the usages of the keywords only.

[1] https://github.com/trending

The tool has been tested on two different datasets, a machine learning taxonomy with 1,600 examples and a bioinformatics one with 876 projects.

LabelGit [232] is a dataset for the classification of Java software projects. The authors proposed an approach to excerpt information directly from source code and dependency graphs. The former is obtained by a combination of techniques like *code2vec* and *fastText*, while the latter is a graph describing the dependencies between classes. Similarly, ClassifyHub [245] is a GitHub projects classification algorithm based on ensemble learning. The approach has been evaluated on a dataset consisting of 681 projects and obtained a precision of about 60% and a recall of 58%.

With Lascad (Language-Agnostic Software Categorization and Similar Application Detection) [16], the authors proposed a tool based on information retrieval techniques, i.e., LDA (Latent Dirichlet Allocation) and hierarchical clustering. First, the tool extracts terms from the dataset's source code, followed by a refining phase where stop words and similar code-specific terms are removed. The second step is the most important one; LDA is applied on the terms corpus to get the topics, then these topics are grouped by using the hierarchical clustering technique. Finally, projects are assigned to each group and then labeled. Based on this result, during the third step, given an application as input, LASCAD retrieves corresponding software ranked by similarity.

Five different machine learning algorithms have been used to classify OSS projects [144], i.e., SVM, Naive Bayesian, Decision Tree, and IBK classifier. The training phase for each model relies on bytecode analysis performed by JClassInfo, a well-founded tool. Each model is fed with API methods, classes, and packages excerpted from 3,286 projects labeled belonging to SourceForge. The results show that SVM outperforms the other models in terms of precision, recall, and success rate.

Wang *et al.* [279] propose a tag recommendation based on a semantic graph (TRG) to classify projects extracted from two OSS communities, i.e., OhLoh and Freecode. The first step is to analyze semantic correlations between the software description and tags and encode them in a hierarchical graph. Then, a list of tags is retrieved given an input project by relying on the constructed graph. TRG eventually ranks the final list of recommendations according to their probability.

An agglomerative hierarchical clustering for taxonomy construction name AHCTC has been proposed [143] to categorize software projects stored on the OhLoh platform. To this end, the approach integrates the technique mentioned above with a topic model based on the well-established LDA algorithm to identify thematic spaces composed of similar tags. Finally, the proposed clustering technique can extract the most central tag from the thematic space to classify the given project properly.

At the time of development, none of the reviewed approaches offers a comprehensive replication package, thus preventing us to compare our first contribution in the domain, i.e., the MNBN approach. Afterward, we enhance the overall performance by developing, HybridRec, the first hybrid RSSE that combines different algorithms for categorizing OSS repositories.

2.1.3 Modeling assistant tools

The Extremo Eclipse plugin [166] supports modeling activities by analyzing information sources obtained from different resources, i.e., Ecore, XMI, RDF, OWL files. Extremo uses such excerpted data to build a common data model by mapping relevant entities. The underpinning query mechanism allows users to customize and refine the retrieved entities by following two styles, i.e., predicate-based or custom.

Dupont *et al.* [77] propose a domain-specific modeling (DSM) environment to assist the specification of models in Papyrus. Given a UML profile, the tool exploits the EMF generator to map each profile metaclass to a concrete Java class. Afterward, the generated Papyrus plugin is used to define custom DSL using a tooling palette to specify graphic components. The proposed environment can be further extended by including different features, e.g., proactive triggering, fine-grain palette customizations, or suggestions to complete the input UML profile, to list a few.

AVIDA-MDE [93] supports the generation of behavioral models starting from the requirements specification by using a digital evolution strategy. Given a well-defined set of parameters and an initial model, the tool elicits instinctual knowledge elements and constructs new transitions to support scenarios and meet the specified constraints. The resultant model is eventually evaluated using a robot navigation system as the testing scenario.

With the aim of supporting the definition of Atom^3 models, Sen *et al.* [238] devise a model assistant based on a constraint logic program (CLP) strategy. The proposed approach induces the complete model using a set of constraints expressed in Prolog. Afterward, the user can customize such generated model by relying on a domain-specific editor. The system eventually solves the specified constraints to produce the final model, which can be decorated with additional elements manually specified by the designer.

The ASketch tool [278] is capable of completing Alloy partial models using automated analysis. Given the input model, the interpreter extracts relevant information using a tailored parser. In such a way, a list of possible abstract candidates is generated and ASketch can find possible solutions by employing an SAT solver. The system eventually fills the model under construction with concrete candidate fragments such that predefined tests, i.e., unit testing, test execution.

Batot and Sahraoui [26] formulate the design of a modeling assistant as a multi-objective optimization problem (MOOP). The system employs the well-known NSGA-II algorithm to search for partial metamodels given a predefined initial set by using the Pareto concept. It helps modelers to complete the delivered models using coverage degrees and pre-defined minimality criteria. Besides textual format specifications, models can be outlined by means of graphical tools, e.g., DIA or Visio. An interactive approach to metamodel construction has been proposed in [152]. Given a model fragment expressed using graphical tools, the system can suggest relevant elements that can be employed to complete the model-under-construction by using well-known refactoring strategies, quality issues for the conceptual scheme, model design patterns, and antipatterns. The system assesses the quality of this initial version against model examples. The validated model is eventually compiled into a given technology, ie EMF or METADEPTH. Similarly, Kuschke *et al.* [137] present a pattern-based approach to recommend relevant completions for structural UML models. Each user operation triggers an event in which the detailed information is detected. The tool retrieves a set of ranked activity candidates (AC) which supports modelers during model editing.

Given an incomplete Simulink model, the SimVMA prototype[249] integrates ML, MDE, and software cloning to provide *(i)* a complete model; or *(ii)* single edit operation on the incomplete model. To this end, the tool employs code cloning techniques and an ML model to identify potential similar candidates by examining past usage statistics of existing models.

Recently, an NLP-based architecture for the autocompletion of partial models has been presented [42]. Given a set of textual documents related to the initial model, relevant terms are extracted to train a contextual model using the basic NLP pipeline, i.e., tokenization, splitting, and stop-word removal. Afterward, the system is fed with the model under construction to automatically slice it following a set of predefined patterns used to search for recommendations. Modelers can give feedback which can be used to update the model and improve the recommendations. A pre-trained neural network was used to recommend relevant metamodel elements, i.e., classes, attributes, and associations [284]. Such data is encoded in tree-based structures that are masked using the well-founded RoBERTa model language architecture plus an NLP pipeline. This preprocessing is needed to obtain an obfuscated training set that are used by the network to produce the outcomes. DoMoBOT [230] combines NLP and a supervised ML model to automatically retrieves domain models from the textual content using the spaCy tool. After the preprocessing phase, the predictive component uses the encoded sentences to retrieve similar model entities and generate the final domain model.

MAR [147] employs a query-by-example approach to search for similar metamodels/-models. First, model structure is encoded as bags of paths before being indexed and stored

on Apache HBase. Given a model, MAR uses it as a query to search for similar artifacts using a similarity score. Though modelers can learn by inspecting similar metamodels, they have to manually examine the results to extract useful information.

In collaborative modeling, Barriga *et al.* [24] propose the adoption of an unsupervised and Reinforcement Learning (RL) approach to repair broken models, which have been corrupted because of conflicting changes. The main intent is to potentially reach model repairing with human-quality without requiring supervision.

To allow developers to design solutions that solve machine learning-based problems by automatically generating code for other technologies as a transparent bridge, a language and a technology-independent development environment are introduced in [87]. A similar tool named OptiML has been proposed [252], aiming to bridge the gap between ML algorithms and heterogeneous hardware to provide a productive programming environment.

Mussbacher *et al.* [170] conducted an initial investigation on Intelligent Modeling Assistants (IMAs) using a comprehensive assessment grid. To elicit critical IMAs features, the authors analyzed the well-founded Reference Framework for Intelligent Modeling Assistance (RF-IMA). The main finding of the work is that existing IMAs obtain low scores in the extracted features and they can be further enhanced in terms of performance as well as in the underpinning structure.

Breuker [40] reviews the main modeling languages used in ML as well as inference algorithms and corresponding software implementations. The aim of this work is to explore the opportunities of defining a DSML for probabilistic modeling.

To support the completion of UML diagrams, a model assistant based on the Doc2Vec strategy has been proposed to recommend domain-specific concepts extracted from source code [45]. First, the approach extracts class diagrams from a curated corpus of Java projects using the reverse engineering strategy. Then, the relevant features are embedded in the Doc2Vec model by exploiting the Glove library to retrieve similar documents given the input context, i.e., a model under construction. The modeler can eventually complete the partial model by using domain-specific concepts extracted from the list of candidate documents.

With respect to the aforementioned approaches, MemoRec is the first tool that applies the context-aware collaborative filtering technique to support metamodeling activities. Furthermore, four different encoding schemes were conceived to enhance the recommendations' capabilities. This dissertation further contributes to the domain by presenting MORGAN, a recommender system based on graph kernels to support the completion of both models and metamodels. Even though it suffers from scalability issues as reported in the dedicated chapter, it was the first approach that employs dedicated parsers to encode three different types of modeling artifacts, i.e., metamodels, models, and JSON schema.

2.2 Democratizing the development of complex systems

2.2.1 Model-driven engineering and low-code development platforms

Over the past decades, several strategies have been proposed to automatize the development of complex systems. Model-driven engineering (MDE) [39] makes use of models as first-class artifacts to facilitate the overall development lifecycle and increase productivity by introducing automation [117, 126]. In this context, a *model* is an abstraction of real world entities that synthesizes their crucial features. Meanwhile, a *metamodel* is a further abstraction that contains statements about the constructs used in models. Roughly speaking, a model *conforms* to a metamodel if the former is built on top of the concepts defined by the latter.

MDE practitioners specify different types of models to simulate, maintain, and generate code for the system. Nonetheless, the application of this paradigm is not limited to code generation. For instance, a particular application domain can be described by relying on Domain-Specific Languages (DSLs), specifically conceived to represent domain-specific entities.

Low-Code Development Platforms (LCDPs) have been recently introduced to simplify the development of fully functional software systems employing advanced graphical user interfaces and visual abstractions requiring minimal or no procedural code [214]. The main goal of LCDPs is to permit users to build their software systems even if they do not have advanced programming skills [281, 225].

A report provided by the Forrest Industry [215] highlights the main LCPD market segments and vendors as well as possible future evolution in this field. Besides the aforementioned benefits and advantages, low-code development requires at least medium level programming skills to properly develop the requested application as well as design process experiences to integrate the different tasks. Concerning the market segments, they are divided by considering the type of application and expected goals i.e., *general-purpose*, *process apps*, *database*, *request-handling*, and *mobile apps*. The majority of them cover the *general-purpose* applications and supports several types of software systems as well as their life-cycle management [226]. Platforms for *process apps* aim to handle collaboration and coordination among different people in very large companies. LCDPs for the *database* segment allow us to handle, manipulate, and retrieve data from relational databases in a user-friendly manner without handling the entire application workflow. *Request-handling* low-code platforms are used to handle custom processes for different departments within a company and to manage requests for services. Similarly to process app development, these platforms must address the integration among various entities and provide self-service functionalities. LCDPs for developing *mobile apps* include strongly dedicated functionalities i.e., mobile middleware

and different widget. Concerning future trends, the segment of the mobile app development will disappear [215]. Recently, LCDPs have been used to support the development of Internet of Things (IoT) applications. In [119], the authors identify a set of features needed to develop IoT applications in a low-code way by reviewing 17 different low-code platforms based on the MDE paradigm. The results show that the examined platforms have some limitations i.e., lack of standards and limited support for testing. Pantelimon *et al.* [198] propose NETIot, a low-code platform to design, configure, and test heterogeneous IoT devices. Even though LCDPs and MDE are close conceptually, not all low-code approaches are model-driven [75].

2.2.2 Supporting the automatic design of recommender system

Elliot [19] provides RS designers with a practical means to design, implement, and evaluate a recommender system. Starting from a configuration file written in YAML, the tool produces the desired system by supporting all the needed phases, i.e., dataset loading, filtering, splitting, or retrieving recommendations. To this end, Elliot offers different models that the user can modify by selecting splitting rules, parameters, and evaluation metrics. The system eventually retrieves the outcomes for each system.

Almonte *et al.* [11] present a generic low-code approach to simplify the development of an RS by employing model-driven engineering. Starting from a predefined metamodel, the system can generate a model and a DSL that expresses relevant concepts to configure the building blocks of an RS. To generate the results, the proposed DSL is built on top of the RankSys external library [265] that provides all the necessary utilities to create, execute and test collaborative filtering RSs. The same authors propose DROID, a model-based tool that makes use of a dedicated DSL to configure and deploy different RSs into modeling frameworks [13]. The first step is the definition of the key operations needed to conceive the system, i.e., candidate recommender methods, the training dataset, and the evaluation metrics. Afterward, the system automatically evaluates each recommendation method using the specified metrics. DROID eventually synthesizes an RS service that can be integrated into different modeling tools.

Recently, the Lavoisier tool has been conceived to automate the data management pipeline typically employed in data analysis applications [59]. By relying on a textual DSL specified through Xtext, the Lavoisier language allows the user to specify the mining algorithm that is used to extract data in a tabular format. To this end, the proposed approach offers high-level primitives to select and rearrange any specific-domain data. In such a way, the low-level data transformation operations are completely handled by Lavoisier that produces tabular datasets ready to be digested by mining algorithms with less effort.

`RECOLIBRY SUITE` [123] drives the user during the design and deployment of a recommender system from scratch by exploiting a set of existing intelligent tools. The design phase is supported by the `RECOLIBRY-CORE` component that offers a catalog of ready-to-use recommender algorithms, i.e., collaborative filtering, machine learning-based strategy. Once the user has selected the system architecture, the deployment phase is realized by using `RECOLIBRY-STUDIO` application that allows the specification of the desired system in terms of input and output. Both the aforementioned intelligent components make use of a common ontology to verify possible constraints on the specified system components. `Reco server` component eventually deploys the system as an external web service that can execute the generated code.

Mettouris *et al.* [161] proposed UbiCARS, a model-driven framework to support the development of Ubiquitous Context-Aware Recommender Systems in the e-commerce domain. First, the authors defined a custom domain-specific modeling language (DSML) to express notable concepts in the context-aware recommender system domain, e.g., user feedback, purchase history, and rating. The system eventually generates seven different datasets for e-commerce and captures implicit feedback by relying on a web interface. Like DROID, UbiCARS is limited to a specific class of recommendation system, i.e., context-aware ones.

A recurrent neural network (RNN) based on the Gated Recurrent Unit (GRU) technique has been used to design a general-purpose recommender system [134]. The approach employs ontologies to represent users' profiles by collecting external knowledge. The extracted data is then used to feed the underpinning model to produce a recommender system that fulfills the user's specification.

Recently, the AutoRec platform [280] has been proposed to automatize the deployment of RSs based on deep neural networks. The system uses TensorFlow to implement a highly flexible pipeline that allows for model selection and hyperparameter tuning. AutoRec exposes all the supported utilities using a user-friendly API to build the designed system. Similarly, Auto-Surprise [17] tool is built on top of the Surprise Python library to automatize the selection and the evaluation of 11 different algorithms implemented by the library, e.g., SVD, KNN-means to name a few. The approach employs a sequential model-based optimization strategy that evaluates the algorithms mentioned above in parallel to select the best one given the initial configuration.

To properly support the user during the specification of the system, such tools needs to be integrated in a proper environment, e.g., IDE, web-based services, or tailored frameworks. Extremo [166] is a recommender system to support modeling activities integrated as an Eclipse plugin that relies on heterogeneous information obtained from different sources. A common data model is devoted to displaying this information in a dedicated repository IDE

view. Users can browse and filter the results employing a search wizard component. Extremo can eventually be used to create DSLs for tailored domains. Similarly, a Papyrus plugin [77] aims to assist modelers in designing domain-specific models that conform to the OMG UML 2 standard. By relying on an EMF generator model, and UML profiling, the deployed plugin can generate the corresponding Java code given the model specification. Furthermore, the tool supports the creation of viewpoints and CSS-style sheets that could be used to customize the generated DSL.

The *Recombee* framework[2] offers a recommender system as a service to support several domains, e.g., multimedia content, job boards, feed aggregators. To enable recommendations, the system uses a set of item catalogs as well as user attributes and real-time interactions. *Recombee* exposes all the supported features as SDKs client libraries for different languages, i.e., Java, Python, Node.js, to name a few. Challenges in designing and deploying code completion IDE plugins have been highlighted in a recent work [290]. The authors conducted an extensive user study and developed a prototype code completion recommender system named tranX-plugin[3] for PyCharm IDE to suggest relevant Python code given a textual query. The outcomes indicate that the current IDE code assistants have some limitations in terms of correctness and quality even though developers provide several suggestions for improvement. Recently, GitHub developed Copilot[4] – an AI-based recommender system deployed as a VS Code extension that analyzes developers' context to suggest relevant code elements, i.e., missing API function calls. To this end, the underpinning AI model has been trained on publicly available source code and natural language content. However, according to the current programming task, the retrieved suggestions need to be eventually adapted by the developer.

Like the approach presented in Chapter 8, namely LEV4REC, the reviewed frameworks employ high-level concepts to configure the system and evaluate it automatically. However, the proposed tool offers a dedicated feature model that drives the developer during the specification of critical phases, i.e., the algorithm selection, producing recommendations, and automatic evaluations.

2.2.3 Automatic machine learning

Recently, the usage of deep neural networks gains attention in several domains, including recommender systems. Even though such techniques come in handy to support several tasks, the overall performance may be hampered by a wrong selection of hyperparameters, which

[2] https://www.recombee.com/
[3] https://github.com/neulab/tranX-plugin
[4] https://copilot.github.com/

constitutes a considerable barrier for non-expert users. The automated machine learning (AutoML) paradigm has been recently proposed to facilitate the design and deployment of ML-based approaches by driving the user in the model selection [118]. AutoML approaches aim at finding the best hyperparameter configuration by formulating the problem as a multi-objective optimization search.

Being built on top of the well-founded Weka ML platform, Auto-Weka [135] exploits Bayesian optimization to find the best model among the available ones. In particular, the tool identifies the combination of Weka's algorithms and their associated parameters that minimizes the cross-validation loss.

Similarly, Auto-Net [159] has been proposed to support the configuration of four types of well-founded neural networks, i.e., multi-layer perceptrons, residual neural networks, shaped multi-layer perceptrons, and shaped residual neural networks. To find the best solution in the considered solution space, the system combines the traditional Bayesian optimization with the bandit-based Hyperband strategy. In such a way, the employed optimization model can find the most promising neural network models by discarding the poor-quality ones in the early phases.

ktrain [154] is a low-code Python library that supports AutoML by relying on various Machine Learning frameworks, e.g., TensorFlow, and Keras. Using the proposed system, a non-expert user can select and customize different models and estimate their learning rate to fine-tune them. At the time of writing, *ktrain* covers text and image classification tasks by including well-known ML datasets and proper preprocessing techniques. Similarly, the *fastai* [113] and Ludwig [165] frameworks support various cutting-edge technologies to speed up the development of an ML application from scratch. *fastai* focuses on the deep learning domain by employing a layered structure composed of high, mid, and low-level Python API. In such a way, end-users with different skill levels can profitably practice complex deep learning utilities. Meanwhile, Ludwig employs a tailored declarative language to design a complete ML workflow. The system can create, optimize, and evaluate several ML off-the-shelf approaches by automatically generating a complete pipeline starting from an initial user specification.

Chapter 3

A conceptual framework to develop RSSEs

The issue of designing and implementing generic recommender systems has been carefully addressed by state-of-the-art studies [207, 220]. Nevertheless, there is a lack of proper references for the design of a recommendation system in a concrete context, i.e., satisfying requirements by various industrial partners. By means of a thorough investigation of the related work, we realized that existing studies tackled the issue of designing and implementing a recommendation system for software engineering in general. However, conceptualizing a generic framework for developing RSSEs is still challenging due to the need to address several issues, e.g., collecting proper training data, or managing the composition of miscellaneous components.

In this chapter, we present a methodological framework that aims at identifying the critical features that are required to engineer RSSEs at the design time. In this respect, we discuss a generic architecture that have been used to develop actual RSSE, including the ones in the context of the EU CROSSMINER project.[1] We exploited cutting-edge information retrieval techniques to build recommender systems, providing software developers with practical advice on various tasks through an Eclipse-based IDE and dedicated analytical Web-based dashboards. Based on the project's mining tools, developers can select open-source software and get real-time recommendations while working on their development tasks. Starting from that, we elicit a feature model that conceptualizes all the features involved in the RSSE development.

Afterward, we elicit a set of metrics that are commonly used to evaluate RSSEs quantitatively. Furthermore, we define a set of concepts that have been used in the evaluation section

[1]https://www.crossminer.org

of each chapter, e.g., the concept of recommended items, true positive and false positive values.

The content of this chapter has been partially adapted from the published manuscript in the Empirical Software Engineering journal [67], by including the additional RSSEs developed during the these three years.

Outline of the chapter: First, Section 3.1 presents a bird-eye architecture that has been elicited from the development systems by discussing the underpinning strategies and technologies employed to build them. Then, we discuss a feature model that aims at supporting the specification of a generic RSSE at the design level in Section 3.2. Afterward, we introduce the metrics that have been used throughout the dissertation to evaluate the different RSSEs in Section 3.3. Section 3.4 eventually concludes the chapter by summarizing the solution architecture and possible application to support additional SE tasks.

3.1 RSSEs bird-eye architecture

Figure 3.1 shows a bird-eye architecture that have been exploited to develop the presented RSSEs. In particular, such a conceptual framework can be used to identify the essential components of a generic RSSEs, each designed to implement the four main activities, which are typically defined for any recommender systems, i.e., data pre-processing, capturing context, producing recommendations, and presenting recommendations [220] as shown in the upper side of Fig. 3.1. Accordingly, the architecture solution is made up of four main modules: the `Data Preprocessing` module contains tools that extract metadata from OSS repositories (see the middle part of Fig. 3.1). Data can be of different types, such as source code, configuration, cross-project relationships, modeling artifacts, or README files. Natural language processing (NLP) tools are also deployed to analyze developer forums and discussions. Furthermore, dedicated solution are needed for some specific domains, e.g., modeling assistants.

The collected data is used to populate a knowledge base which serves as the core for the mining functionalities. By capturing developers' activities (`Capturing Context`), an IDE is able to generate and display recommendations (`Producing Recommendations` and `Presenting Recommendations`). In particular, the developer context is used as a query sent to the knowledge base that answers with recommendations that are relevant to the developer contexts (see the lower side of Fig. 3.1). Machine learning techniques are used to infer knowledge underpinning the creation of relevant real-time recommendations.

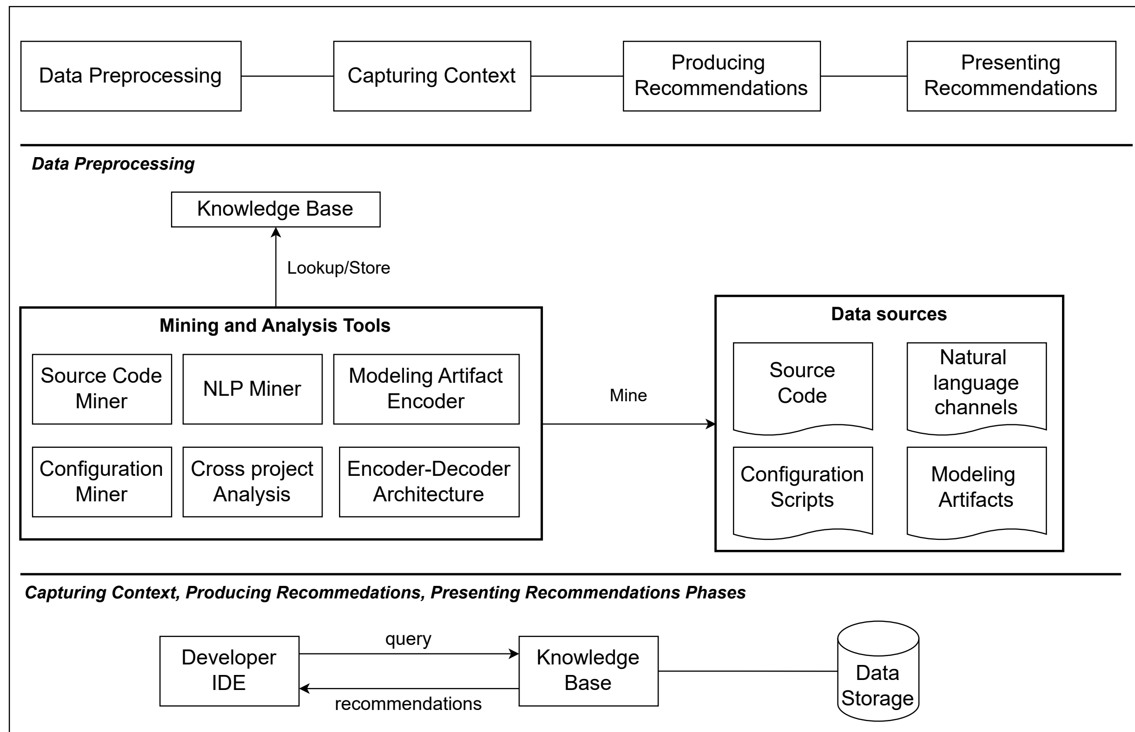


Fig. 3.1 Bird-eye architecture of developed RSSEs.

The RSSEs `knowledge base` allows developers to gain insights into raw data produced by different mining tools, which are the following ones:

- *Source code miners* to extract and store actionable knowledge from the source code of a collection of open-source projects;
- *NLP miners* to extract quality metrics related to the communication channels, and bug tracking systems of OSS projects by using Natural Language Processing and text mining techniques;
- *Configuration miners* to gather and analyze system configuration artifacts and data to provide an integrated DevOps-level view of a considered open source project;
- *Cross-project miners* to infer cross-project relationships and additional knowledge underpinning the provision of real-time recommendations;
- *Modeling Artifact Encoder* to infer mutual relationships among the considered modeling artifacts, i.e., metamodels, models, and JSON schema;
- *Encoder-Decoder Architecture* to provide cohesive API function calls considering the temporal sequences.

Table 3.1 The examined use cases.

No.	Artifact	Description	Developed tool
①	*Similar OSS projects*	We crawl data from OSS platforms to find similar projects to the system being developed, with respect to different criteria, e.g., external dependencies, or API usage [183]. This type of recommendation is beneficial to the development since it helps developer learn how similar projects are implemented.	CrossSim
②	*Additional components*	During the development phase, programmers search for components that projects similar to the one under developed have also included, for instance, a list of external libraries [157], or API function calls [167].	CrossRec, FOCUS
③	*Code snippets*	Well-defined snippets showing how an API is used in practice are extremely useful. These snippets provide developers with a deeper insight into the usage of the APIs being included [181].	FOCUS, LUPE
④	*Relevant topics*	GitHub uses tags as a means to narrow down the search scope. The goal is to help developers approach repositories, and thus increasing the possibility of contributing to their development and widespread their usage [72, 70].	MNBN, HybridRec
⑤	*Relevant modeling elements*	Model-driven engineering (MDE) exploits models as first-class artifacts to develop a plethora of software application. In this respect, it is crucial to assist modelers during the specification phase [69, 76].	MemoRec, MORGAN

Part of the recommender systems have been developed under the umbrella of CROSSMINER to satisfy the requirements by six industrial use-case partners of the project working on different domains including IoT, multi-sector IT services, API co-evolution, software analytics, software quality assurance, and OSS forges.[2]. In this respect, we moved a step forward by covering additional use cases and domain, e.g., modeling assistants. In particular, Table 3.1 specifies the considered application domain solicited both by CROSSMINER partner and academic community. To satisfy the given requirements the following recommender systems have been developed:

- *CrossSim* [175, 183] – It is an approach for recommending similar projects with respect to the third-party library usage, stargazers and committers, given a specific project;
- *CrossRec* [182] – It is a framework that makes use of **C**ross Projects **R**elationships among **O**pen **S**ource **S**oftware Repositories to build a library **Rec**ommendation System on top of CrossSim;
- *FOCUS* [181, 177] – The system assists developers by providing them with API function calls and source code snippets that are relevant for the current development context;

[2]https://www.crossminer.org/consortium

- *LUPE* [190] – Similar to FOCUS, this approach suggests API function calls and code snippets by considering the active context. Nevertheless, LUPE is able to provide cohesive recommendations by exploiting the underpinning encoder-decoder architecture;
- *MNBN* [72] – It is an approach based on a Multinomial Naive Bayesian network technique to automatically recommend topics given the README file(s) of an input repository;
- *HybridRec* [70] – Build on top of MNBN, this approach combines a refined stochastic model with collaborative filtering technique to increase the coverage of recommended GitHub topics;
- *MemoRec* [69] – Given a partial metamodel, this approach can provide relevant attributes and structural features by relying on dedicated encoding schemes;
- *MORGAN* [68, 76] – Similar to MemoRec, the tool is capable of supporting model completion using graph kernel similarity technique. In addition, a set of dedicated parsers allow MORGAN handling additional model artifacts besides metamodel, i.e., class models, and JSON schema.

It is worth noting that two of the presented tools is not part of this dissertation, i.e., CrossRec and CrossSim. Nonetheless, we keep them as the purpose of this chapter to present the general principles that have been used in the CROSSMINER project.

By referring to Fig. 3.1, the developed recommender systems are implemented in the `Knowledge Base` component. Moreover, it is important to remark that even though such tools can be used in an integrated manner directly from the `Developer IDE`, their combined usage is not mandatory. They are different services that developers can even use separately according to their needs.

3.2 Main RSSE design features

Being aware of existing systems is also important to elicit common features at design time. For studying a recommendation system, besides conventional quantitative and qualitative evaluations, it is necessary to compare it with state-of-the-art approaches. Such an issue is also critical in other domains, e.g., Linked Data [194], or music recommendations [178, 236].

To this end, by analyzing the existing literature about recommendation systems and the developed RSSEs presented in the previous section we identified and modeled their relevant variabilities and commonalities. Our results are documented using feature diagrams

which are a common notation in domain analysis [56]. Figure 3.2 shows the top-level features or recommender systems, i.e., *Data Preprocessing*, *Capturing Context*, *Producing Recommendations*, and *Presenting Recommendations* in line with the main functionalities typically implemented by recommender systems. We extracted all the shown components mainly from existing studies [34, 138, 220] as well as from our development experience under the needs of the CROSSMINER project. The top-level features shown in Fig. 3.2 are described below.

Data Preprocessing: In this phase techniques and tools are applied to extract valuable information from different data sources according to their nature. In particular, *structured data* adheres to several rules that organize elements in a well-defined manner. Source code and XML documents are examples of this category. Contrariwise, *unstructured data* may represent different content without defining a methodology to access the data. Documentation, blog, and plain text fall into this category. Thus, the data preprocessing component must be carefully chosen considering the features of these miscellaneous sources.

`ASTParsing` involves the analysis of structured data, typically the source code of a given software project. Several libraries and tools are available to properly perform operations on ASTs, e.g., fetching function calls, retrieving the employed variables, and analyzing the source code dependencies. Additionally, snippets of code can be analyzed using `Fingerprints`, i.e., a technique that maps every string to a unique sequence of bits. Such a strategy is useful to uniquely identify the input data and compute several operations on it, i.e., detect code plagiarism as shown in [297].

Moving to unstructured input, `Tensors` can encode mutual relationships among data, typically users' preferences. Such a feature is commonly exploited by collaborative filtering approaches as well as by heavy computation on the input data to produce recommendations. Plain text is the most spread type of unstructured data and it includes heterogeneous content, i.e., API documentation, repository's description, Q&A posts, to mention a few. A real challenge is to extract valuable elements without losing any relevant information. Natural processing language (`NLP`) techniques are employed to perform this task by means of both syntactic and semantic analysis. Stemming, lemmatization, and tokenization are the main strategies successfully applied in existing recommender systems. Even the *MNBN* approach previously presented employs such techniques as preparatory task before the training phase. Similarly to tensors, `GraphRepresentation` is useful to model reciprocal associations among considered elements. Furthermore, graph-based data encodings can be used to find peculiar patterns considering nodes and edges semantic.

Capturing Context: After the data preprocessing phase, the developer context is excerpted from the programming environment to enable the underpinning recommendation engine. A

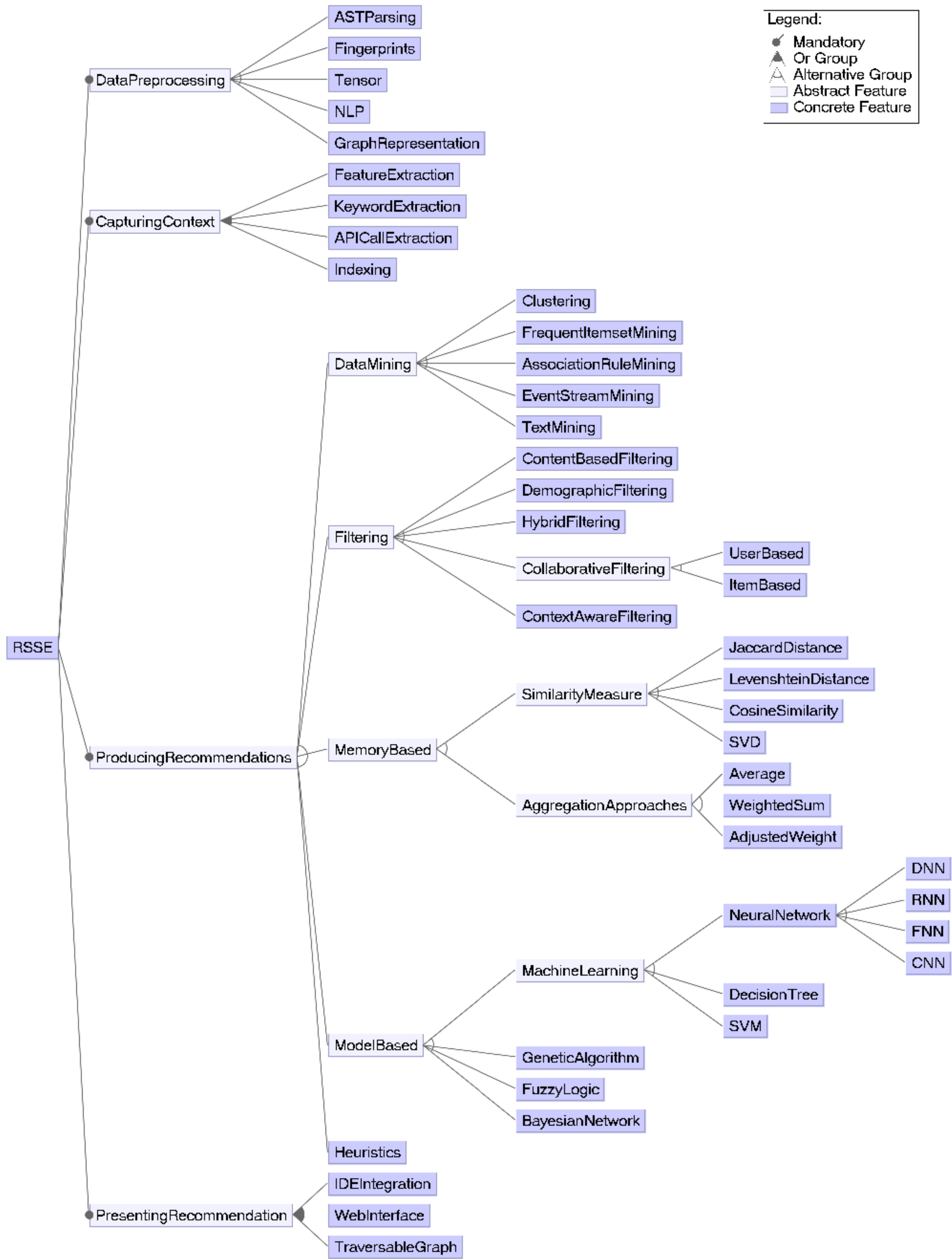


Fig. 3.2 Main design features of recommendation systems in software engineering.

well-founded technique primarily employed in the ML domain is the FeatureExtraction to concisely represent the developer's context. Principal Component Analysis (PCA) and Latent Semantic analysis (LDA) are just two of such techniques employed for such a purpose. Keyword extraction and APICallExtraction are two techniques mostly used when the *Capturing Context* phase has to analyze source code. Capturing context often involves the search over big software projects. Thus, a way to store and access a large amount of data is necessary to speed up the recommendation item delivery. Indexing is a technique mainly used by the code search engines to retrieve relevant elements in a short time.

Producing Recommendations: In this phase, the actual recommendation algorithms are chosen and executed to produce suggestions that are relevant for the user context, once it is previously captured. By several variating parameters such as type of the required input and the underlying structure, we can elicit different features as represented in the diagram shown in Fig. 3.2. Concerning Data Mining techniques, some of them are based on pattern detection algorithms, i.e., Clustering, FrequentItemsetMining, and AssociationRuleMining. Clustering is usually applied to group objects according to some similarity functions. The most common algorithm is the K-means based on minimizing the distance among the items. A most representative element called centroid is calculated through a linkage function. After such a computation, this algorithm can represent a group of elements by referring to the most representative value. FrequentItemsetMining aims to group items with the same frequencies, whereas AssociationRuleMining uses a set of rules to discover possible semantic relationships among the analysed elements. Similarly, the EventStreamMining technique aims to find recurrent patterns in data streams. A stream is defined as a sequence of events usually represented by a Markov chain. Through this model, the algorithm can exploit the probability of each event to establish relationships and predict a specific pattern. Finally, TextMining techniques often involve information retrieval concepts such as entropy, latent semantic analysis (LSA), or extended boolean model. In the context of producing recommendations, such strategies can be used to find similar terms by exploiting different probabilistic models that analyze the correlation among textual documents.

The availability of users' preferences can affect the choice of recommendation algorithms. Filtering strategies dramatically exploit the user data, e.g., their ratings assigned to purchased products. ContentBasedFiltering (CBF) employs historical data referring to items with positive ratings. It is based on the assumption that items with similar features have the same score. Enabling this kind of filtering requires the extraction of the item attributes as the initial step. Then, CBF compares the set of active items, namely the context, with possible similar items using a similarity function to detect the closer ones to the user's needs. DemographicFiltering compares attributes coming from the users themselves instead of the

purchased items. These two techniques can be combined in `HybridFiltering` techinques to achieve better results.

So far, we have analyzed filtering techniques that tackle the features of items and users. `CollaborativeFiltering` (CF) approaches analyze the user's behaviour directly through its interaction with the system, i.e., the rating activity. `UserBased` CF relies on explicit feedback coming from the users even though this approach suffers from scalability issues in case of extensive data. The `ItemBased` CF technique solves this issue by exploiting users' ratings to compute the item similarity. Finally, `ContextAwareFiltering` involves information coming from the environment, i.e., temperature, geolocalization, and time, to name a few. Though this kind of filtering goes beyond the software engineering domain, we list it to complete the filtering approaches landscape.

The `MemoryBased` approach acts typically on user-item matrixes to compute their distance involving two different methodologies, i.e., `SimilarityMeasure` and `AggregatationApproach`. The former involves the evaluation of the matrix similarity using various concepts of similarity. For instance, `JaccardDistance` measures the similarity of two sets of items based on common elements, whereas the `LevenshteinDistance` is based on the edit distance between two strings. Similarly, the `CosineSimilarity` measures the euclidean distance between two elements. Besides the concept of similarity, techniques based on matrix factorization are employed to make the recommendation engine more scalable. Singular value decomposition (`SVD`) is a technique being able to reduce the dimension of the matrix and summarize its features. Such a strategy is used to cope with a large amount of data, even though it is computationally expensive. `AggregatationApproach`es analyze relevant statistical information of the dataset such as the variance, mean, and the least square. To mitigate bias lead by the noise in the data, the computation of such indexes use adjusted weigths as a coefficient to rescale the results.

To produce the expected outcomes, `MemoryBased` approaches require the direct usage of the input data that cannot be available under certain circumstances. Thus, `ModelBased` strategies can overcome this limit by generating a model from the data itself. `MachineLearning` offers several models that can support the recommendation activity. `NeuralNetwork` models can learn a set of features and recognize items after a training phase. By exploiting different layers of neurons, the input elements are labeled with different weights. Such values are recomputed during different training rounds in which the model learns how to classify each element according to a predefined loss function. Depending on the number of layers, the internal structure of the network, and other parameters, it is possible to use different kinds of neural networks including Deep Neural Networks (`DNN`), Recurrent Neural Networks (`RNN`), Feed-forward Neural Networks (`FNN`), or Convolutional Neural Networks (`CNN`). Besides ML

models, a recommendation system can employ several models to suggest relevant items. `GeneticAlgorithms` are based on evolutionary principles that hold in the biology domain, i.e., natural species selection. `FuzzyLogic` relies on a logic model that extends classical boolean operators using continuous variables. In this way, this model can represent the real situation accurately. Several probabilistic models can be used in a recommendation system. `BayesianNetwork` is mostly employed to classify unlabeled data, although it is possible to employ it in recommendation activities.

Besides all these well-founded techniques, recommended items can be produced by means of `Heuristics` techniques to encode the knowhow of domain experts. Heuristics employ different approaches and techniques together to obtain better results as well as to overcome the limitations of other techniques. On the one hand, heuristics are easy to implement as they do not rely on a complex structure. On the other hand, they may produce results that are sub-optimal compared to more sophisticated techniques.

3.3 Evaluation metrics for RSSE

Once an RSSE has been developed, there is the need to evaluate the overall performance by adopting a set of standard evaluation strategies. In the following, we will present a set of definitions and metrics that are widely adopted in the community to evaluate those complex systems.

3.3.1 Terminology

The *recommendation outcome* is normally a ranked list of items, e.g., third-party libraries [182, 259], API calls [167, 181], or GitHub topics [72]. Normally, a developer pays attention only to the *top-N* items. Thus, by comparing the items in the ranked list with those stored as ground-truth data, we can examine how well the recommendation system performs. There are various metrics to analyze the performance of a recommendation system. To our knowledge, several studies in RSSE focus only on accuracy [41, 83, 158, 259]. However, while developing the abovementioned RSSEs, we realized that while accuracy is a good metric for evaluating an RS, it is not enough for studying all the performance traits of the outcomes, as it is the case with conventional recommendation systems [90]. As a result, other metrics should also be incorporated to analyze various quality aspects as they are presented as follows. To compute such metrics, the following notations are defined:

- N is the cut-off value for the list of recommended items;

- for a testing project p, the ground-truth dataset is named as *GT(p)*;
- $REC(p)$ is the *top-N* items, it is a ranked list in descending order of real scores, with $REC_r(p)$ being the item in the position r;
- if a recommended item $i \in REC(p)$ for a testing project p is found in the ground truth of p, i.e., *GT(p)*, hereafter we call this as a *match* or *hit*.
- *True positive (Tp)*: the items that are correctly recommended;
- *False positive (Fp)*: the recommended items which actually do not belong to the ground-truth data;
- *False negative (Fn)*: the items that should be present in the recommended ones, but they are not.

3.3.2 Metrics

The metrics that have been employed for evaluating the recommender systems are explained below.

Success rate. Given a set of P testing projects, this metric measures the rate at which a system can return at least a match among *top-N* recommended items for every project $p \in P$ [259]. It is formally defined as follows:

$$success\ rate@N = \frac{count_{p \in P}(\left|GT(p) \bigcap (\cup_{r=1}^{N} REC_r(p))\right| > 0)}{|P|} \tag{3.1}$$

where the function *count()* counts the number of times that the boolean expression specified in its parameter is *true*.

Accuracy. Given a list of *top-N* items, *precision@N*, *recall@N*, and *normalized discounted cumulative gain (nDCG)* are utilized to measure the *accuracy* of the recommendation results.

Precision@N is the ratio of the *top-N* recommended items belonging to the ground-truth dataset:

$$precision@N(p) = \frac{\sum_{r=1}^{N} |GT(p) \bigcap REC_r(p)|}{N} \tag{3.2}$$

Recall@N is the ratio of the ground-truth items appearing in the N items [58, 179]:

$$recall@N(p) = \frac{\sum_{r=1}^{N} |GT(p) \bigcap REC_r(p)|}{|GT(p)|} \tag{3.3}$$

nDCG: Precision and recall reflect well the accuracy, however they neglect ranking sensitivity [28]. nDCG is an effective way to measure if a system can present highly relevant items on the top of the list:

$$nDCG@N(p) = \frac{1}{iDCG} \cdot \sum_{i=1}^{N} \frac{2^{rel(p,i)}}{log_2(i+1)} \tag{3.4}$$

where iDCG is used to normalize the metric to 1 when an ideal ranking is reached.

<u>*TopRank.*</u> It measures the percentage of the first top elements in the ground-truth data:

$$Top\ rank = \frac{TpRank(r)}{|R|} \times 100\% \tag{3.5}$$

where *TpRank(r)* returns 1 if the first predicted element belongs to *UsrTp(r)*, 0 otherwise.

<u>*Sales Diversity.*</u> In merchandising systems, *sales diversity* is the ability to distribute the products across several customers [179, 269]. In the context of mining software repositories, sales diversity means the ability of the system to suggest to projects as much items, e.g., libraries, code snippets, as possible, as well as to spread the concentration among all the items, instead of presenting a specific set of them [220].
Catalog coverage measures the percentage of items recommended to projects:

$$coverage@N = \frac{\left|\cup_{p\in P} \cup_{r=1}^{N} REC_r(p)\right|}{|I|} \tag{3.6}$$

where I is the set of all items available for recommendation and P is the set of projects.
Entropy evaluates if the recommendations are concentrated on only a small set or spread across a wide range of items:

$$entropy = -\sum_{i\in I} \left(\frac{\#rec(i)}{total}\right) ln \left(\frac{\#rec(i)}{total}\right) \tag{3.7}$$

where $\#rec(i) = count_{p\in P}(\left|(\cup_{r=1}^{N} REC_r(p)) \ni i\right|)$, $(i \in I)$ is the number of projects getting i as a recommendation, *total* denotes the total number of recommended items across all projects.

<u>*Novelty.*</u> The metric gauges if a system is able to expose items to projects. *Expected popularity complement* (EPC) is utilized to measure *novelty* and is defined as follows [268, 269]:

$$EPC@N = \frac{\sum_{p\in P} \sum_{r=1}^{N} \frac{rel(p,r)*[1-pop(REC_r(p))]}{log_2(r+1)}}{\sum_{p\in P} \sum_{r=1}^{N} \frac{rel(p,r)}{log_2(r+1)}} \tag{3.8}$$

where $rel(p,r) = |GT(p) \bigcap REC_r(p)|$ represents the relevance of the item at the r position of the *top-N* list to project p; $pop(REC_r(p))$ is the popularity of the item at the position r in the *top-N* recommended list. It is computed as the ratio between the number of projects that receive $REC_r(p)$ as recommendation over the number of projects that are recommended items among the most often recommended ones. Equation 3.8 implies that the more unpopular items a system recommends, the higher the EPC value it obtains and vice versa.

Confidence. Given a pair of <*query, retrieved item*> confidence is the score the evaluator assigns to the similarity between the two items;

Ranking. In a ranked list, it is necessary to have a good correlation with the scores given by the human evaluation [41]. The Spearman's rank correlation coefficient r_s [247] is used to measure how well a similarity metric ranks the retrieved projects given a query. Considering two ranked variables $r_1 = (\rho_1, \rho_2, .., \rho_n)$ and $r_2 = (\sigma_1, \sigma_2, .., \sigma_n)$, r_s is defined as: $r_s = 1 - \frac{6\sum_{i=1}^{n}(\rho_i - \sigma_i)^2}{n(n^2-1)}$. We also employed *Kendall's tau* coefficient [127], which is used to measure the ordinal association between two considered quantities. Both r_s and τ range from -1 (perfect negative correlation) to +1 (perfect positive correlation); $r_s = 0$ or $\tau = 0$ implies that the two variables are not correlated.

Levenshtein Distance. The aforementioned metrics do not take into account the order of the recommended items. This is important during the development phase, as developers need to know the position where they have to insert the software element. Thus, we rely on distance metrics to measure the similarity that takes into account the order of appearance. Given two strings s_1 and s_2, the Levenshtein edit distance between them corresponds to the number of substitutions performed to transform s_1 to s_2. The metric is defined as follows.[3]

$$L_{s_1,s_2}(i,j) = \begin{cases} max(i,j) & \text{if min(i,j)=0,} \\ min \begin{cases} L_{s_1,s_2}(i-1,j)+1 \\ L_{s_1,s_2}(i,j-1)+1 \\ L_{s_1,s_2}(i-1,j+1)+1 \end{cases} & \text{otherwise.} \end{cases} \tag{3.9}$$

where i and j are the terminal character position of strings s_1 and s_2, respectively.

Recommendation time. Being able to provide recommended items in a limited amount of time is important, especially for applications that require instant recommendations. This metric evaluates the duration of time, starting from when a user sends the query until the final recommendations are returned.

[3] https://dzone.com/articles/the-levenshtein-algorithm-1

Depending on the context, we have to choose a suitable set of metrics to evaluate a recommendation system. For example, with CrossSim we can only make use of *Success rate*, *Confidence*, *Precision*, *Ranking*, and *Execution time* to evaluate the tool, since we relied on a user study. Meanwhile, with HybridRec or FOCUS, since we can use the ten-fold cross validation technique (i.e., by exploiting the testing data which was already split into query and ground-truth data), we evaluated them using *Accuracy*, *Precision*, *Recall*, *Diversity*, and *Novelty*.

3.4 Conclusion

In this chapter, we presented a conceptual framework to support the development of RSSEs in terms of techniques, strategy, and high-level features. First, we discussed a generic RSSEs architecture that has been elicited by relying on the gained experience. Such a framework covers all the phases of specifying an RSSE, spanning from the identification of critical building blocks to the actual development. We presented a set of RSSEs that support different SE tasks, i.e., suggesting reusable software artifacts, categorizing OSS projects, and supporting modeling activities. Even though some of them have been conceived during the CROSSMINER project, some elicited concepts hold for generic RSSEs. In addition, we elicited a feature model starting from the developed approaches that can embrace all the critical components needed to support the specification of this class of systems. We eventually discussed a set of widely adopted metrics that have been used to evaluate the CROSSMINER RSSEs quantitatively. It is worth mentioning that all the foundational aspects presented in this chapter have been used to conceive and evaluate the developed systems in this dissertation.

Chapter 4

Recommending API function calls to support developers

When dealing with certain programming tasks, rather than implementing new systems from scratch, developers often make use of third-party libraries that provide the desired functionalities. Such libraries expose their functionality through Application Programming Interfaces (APIs) which govern the interaction between a client project and its incorporated libraries. To use a library in a proper way, it is necessary to use the correct sequence of API calls, known as API usage patterns. The knowledge needed to manipulate an API can be extracted from various sources: the API source code itself, the official website and documentation, Q&A websites such as StackOverflow, forums and mailing lists, bug trackers, other projects using the same API, etc. However, official documentation often merely reports the API description without providing non-trivial example usages. Besides, querying informal sources such as StackOverflow might become time-consuming and error-prone [218]. Also, API documentation may be ambiguous, incomplete, or erroneous [263], while API examples found on Q&A websites may be of poor quality [173]. In this respect, we come across the following motivating question:

> *Which API calls should this piece of code invoke, given that it has already invoked these API calls?*

The problem of recommending API function calls and usage patterns has garnered considerable efforts and attention of the research community in recent years [298, 167]. Several techniques have been developed to automate the extraction of API *usage patterns* [219] for reducing developers' burden when manually searching these sources, and for providing them with high-quality code examples. However, these techniques, based on clustering [276, 298]

or predictive modeling [83], still suffer from high redundancy and poor run-time performance. Moreover, most of the existing approaches are based on clustering on data from code snippets to recommend API usage, which sustains redundancy. In an attempt to transcend the limitations, this chapter presents two different approaches, i.e., FOCUS and LUPE, that rely on context-aware collaborative filtering and deep neural network respectively.

FOCUS [177] mines open-source software repositories to provide developers with API *FunctiOn Calls and USage patterns*. We aim to suggest to developers highly relevant API usages that ease the development process. Our tool distinguishes itself from other tools that recommend API usages as it can provide both function calls and real code snippets that match well with the developer's context. By considering API methods as products and client code as customers, we reformulate the problem of usage pattern recommendation in terms of a collaborative-filtering recommender system. We modeled the mutual relationships among projects using a tensor and mined API usage from the most similar projects. An empirical evaluation has been conducted on a large number of Java projects extracted from GitHub and the Maven Central repository to study FOCUS's performance, and to compare it with a state-of-the-art tool for API usage patterns mining, i.e., PAM [83]. We simulated different stages of a development process, by removing portions of client code and assessing how FOCUS can recommend snippets with API invocations to complete the code being developed. The experiments showed that FOCUS outperforms PAM, with regards to success rate, accuracy, and execution time. Furthermore, we assess the FOCUS's capability of assisting mobile developers by means of an Android dataset. In this chapter, we present the extended version that has been published in IEEE Transactions on Software Engineering [177], that further advances the original tool presented in a conference paper [181]. The candidate contributes in *(i)* developing the data extraction component; and *(ii)* participating in the user study.

The second part of the chapter is devoted to LUPE [190], a deep **L**earning seq**U**ence-to-sequence based system to provide a**P**i r**E**commendations.[1] While most of the existing approaches [83, 167, 177, 192, 276, 298] recommend a ranked list of independent APIs, LUPE *goes one step further to provide a sequence of cohesive API calls*, thanks to the underpinning sequence-to-sequence deep learning algorithms. The proposed system has been evaluated using two real-world Android datasets collected from Google Play and GitHub. The experimental results show that LUPE retrieves a perfect match for several testing method definitions.

Although FOCUS has been conceived to address the same problem, we didn't compare it with LUPE since the two approaches are different by construction, thus leading to unfair

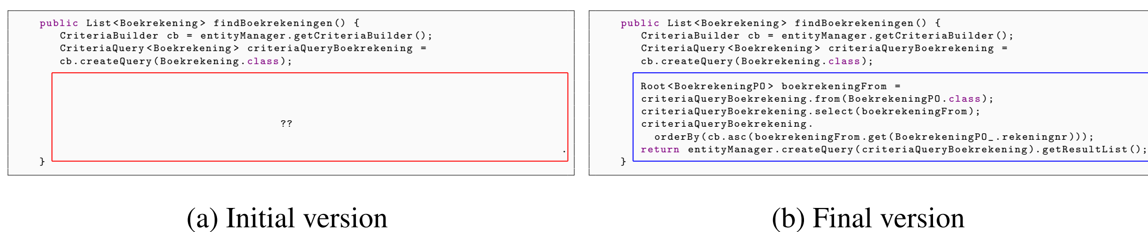
[1]"Lupe" in some languages means magnifying glass, a tool that people normally use when searching for and fitting tiny puzzles.

comparison. LUPE has been published in the Expert System with Applications journal [190] and the candidate has contributed to running and devising the comparison study with the two baselines.

Outline of the chapter: A motivating example and the background technologies related to FOCUS have been presented in Section 4.1.1. Section 4.1.2 gives an overview of the FOCUS architecture and Section 4.1.3 presents the evaluation methodology. We discuss the obtained results and possible issues in Section 4.1.4 and Section 4.1.5 respectively. The second part of the chapter starts with Section 4.2.1, where the technological background of LUPE has been presented. The employed encoder-decoder architecture is described in Section 4.2.2. We summarize the datasets and the adopted evaluation strategies in Section 4.2.3 and assess the LUPE capabilities in section 4.2.4. We discuss the limitations of our work in Section 4.2.5. Section 4.3 summarize the contributions of the chapter and discuss possible future work in the domain.

4.1 FOCUS

4.1.1 Motivation and background



(a) Initial version (b) Final version

Fig. 4.1 The FOCUS motivating example.

In Figure 4.1a, we simulate the behavior of a programmer who is developing a method definition,[2] which is incomplete at the time of consideration. The definition is used to send files to a dedicated server through TCP connections. There are two frames; while the lower frame is left blank, representing missing code, the upper one is already filled with written code. First, a socket is initialized with the `Socket()` API, and the `sock.getInputStream()` method is then used to initialize an input stream. Two variables are instantiated with `FileOutputStream()` and `BufferedOutputStream()`. Afterwards, `is.read()` gets the next bytes from the input stream.

[2]The code example is extracted from the following link: https://bit.ly/2ZrbTaA. We omitted some variable declarations for clarity.

The typical setting considered by FOCUS is shown in Fig. 4.1a: a programmer is implementing some methods to satisfy the requirements of the system being developed. The development is at an early stage, and the developer already used some methods of the chosen API to realize the required functionality. However, she is not sure how to proceed from this point. Under such circumstances, the programmer may browse different sources of information, including Stack Overflow, video tutorials, official API documentation, ifnextchar.etcetc..

Figure 4.1b depicts the final version of the snippet in Fig. 4.1a. In the framed code, the *findBoekrekeningen* method queries the available entities and retrieves those of type *Boekrekening*. To this end, the *Criteria API* library[3] is used as it provides useful interfaces for querying system entities according to the defined criteria. FOCUS has been conceptualized to do the exactly same thing: it is able to suggest to developers recommendations consisting of a list of API method calls that should be used next. Furthermore, it also recommends real code snippets that can be used as a reference to support developers in finalizing the method definition under development.

4.1.1.1 Terminology

"*Collaborative Filtering* (CF) is the process of filtering or evaluating items through the opinions of other people" [235]. In a CF system, a *user* who buys or uses an *item* attributes a rating to it based on her experience and perceived value. Therefore, a *rating* is the association of a user and an item through a value in a given unit (usually in scalar, binary, or unary form). The set of all ratings of a given user is also known as a *user profile* [49]. Moreover, the set of all ratings given in a system by existing users can be represented in a so-called *rating matrix*, where a row represents a user, and a column represents an item.

The expected outcome of a CF system is a set of predicted ratings (aka. *recommendations*) for a specific user and a subset of items [235]. The recommender system considers the most similar users (aka. *neighbors*) to the *active* user to suggest new ratings. A similarity function $sim_{usr}(u_a, u_j)$ computes the *weight* of the active user profile u_a against each of the user profiles u_j in the system. Finally, to suggest a recommendation for an item i based on this subset of similar profiles, the CF system computes a weighted average $r(u_a, i)$ of the existing ratings, where $r(u_a, i)$ varies with the value of $sim_{usr}(u_a, u_j)$ obtained for all neighbors [49, 235].

Context-aware CF systems compute recommendations based not only on neighbors' profiles but also on the *context* where the recommendation is demanded. Each rating is associated with a context [49]. Therefore, for a tuple C modeling different contexts, a

[3] https://docs.oracle.com/javaee/6/tutorial/doc/gjivm.html

context similarity metric $sim_{ctx}(c_a, c_i)$, for $c_a, c_i \in C$ is computed to identify relevant ratings according to a given context. Then, the weighted average is reformulated as $r(u_a, i, c_a)$ [49].

Furthermore, we define the following concepts:

- **APIs**. In software development, an *API* represents a code unit offering pieces of reusable functionality. Developers can use APIs following a precisely defined interface without understanding the API's internal design. Therefore, an API works as a black box, allowing for reuse and modularity [199, 218].
- **method definition** (or simply **definition**) consists of a name, a list of parameter types, a return type, and a body. A definition body may invoke several API function calls, as for instance in Figure 4.1a, the `main(String[] args)` definition calls the `Socket.getInputStream()` API.
- Each API contains public methods M and fields F available to clients. For instance, Figure 4.1b shows the invocation of the method `getInputStream()` defined in the class `java.net.Socket` (see Line 13). An **API invocation** is a call made by a method definition $d \in D$ to another method $m \in M$.

4.1.2 FOCUS architecture

4.1.2.1 Input data

A *software project* is a standalone source code unit that performs a set of tasks. Furthermore, an *API* is like a black-box, i.e., an interface that abstracts the piece of functionality offered by a project by hiding its implementation details. This interface is meant to support reuse and modularity [199, 218]. An API X built in an object-oriented programming language, e.g., the *Criteria API* in Fig. 4.1a, consists of a set T_X of public types, e.g., *CriteriaBuilder* and *CriteriaQuery*. Each type in T_X consists of a set of public methods and fields that are available to client projects, e.g., the method *createQuery* of the type *CriteriaQuery*.

A *method declaration* consists of a name, a (possibly empty) list of parameters, a return type, and a (possibly empty) body, e.g., the method *findBoekrekeningen* in Fig. 4.1b. Given a set of declarations D in a project P, an API *method invocation* i is a call made from a declaration $d \in D$ to another declaration m. Similarly, an API *field access* is an access to a field $f \in F$ from a declaration d in P. API method invocations MI and field accesses FA in P form the set of API usages $U = MI \cup FA$. Finally, an *API usage pattern* (or code snippet) is a sequence $(u_1, u_2, ..., u_n)$, $\forall u_k \in U$. For the sake of presentation, in the scope of this chapter the following terms are used interchangeably: *method declaration* vs. *declaration*

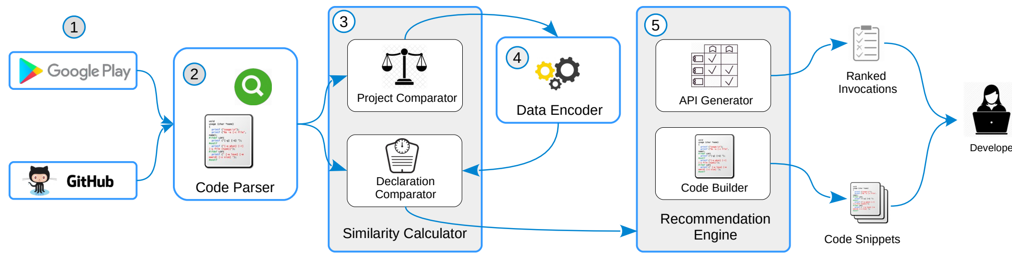


Fig. 4.2 Overview of the FOCUS architecture.

and *API* vs. *invocation*. For each declaration, we extract its method name, a list of types of the parameters, and a list of API function calls. In this way, a project is represented as a set of declarations from its constituent classes.

FOCUS makes use of a context-aware collaborative-filtering technique to search for invocations from highly relevant projects. This allows us to consider both project and declaration similarities to recommend APIs and code snippets. Following the terminology of recommender systems [49], we treat *projects* as the enclosing *contexts*, *method declarations* as *users*, and *method invocations* as *items*. Intuitively, we recommend a method invocation for a declaration in a given project, which is analogous to recommending an item to a customer in a specific context. For instance, the set of method invocations and the usage pattern (cf. framed code in Fig. 4.1b) recommended for the declaration `findBoekrekeningen` can be obtained from a set of similar projects and declarations in a codebase. The *collaborative* aspect of the approach enables to extract recommendations from the most similar projects, while the *context-awareness* aspect enables to narrow down the search space further to similar declarations.

4.1.2.2 The FOCUS engine

The architecture of FOCUS is depicted in Fig. 4.2. To provide its recommendations, FOCUS considers a set of *OSS Repositories* ①. The *Code Parser* ② component extracts method declarations and invocations from the source code or bytecode of these projects. *Project Comparator*, a subcomponent of *Similarity Calculator* ③, measures the similarity between projects in the repositories and the project under development. Using the set of projects and the information extracted by *Code Parser*, the *Data Encoder* ④ component computes rating matrices which are introduced later in this section. Afterwards, *Declaration Comparator* computes the similarities between declarations. From the similarity scores, *Recommendation Engine* ⑤ generates recommendations, either as a ranked list of API function calls using *API Generator*, or as usage patterns using *Code Builder*, which are presented to the developer. In the remainder of this section, we present in greater details each of these components.

4.1.2.3 Code Parser

FOCUS is dependent on Rascal M^3 [25] to function. Rascal M^3 is an intermediate model that performs static analysis on source code to extract method declarations and invocations from a set of projects. This model is an extensible and composable algebraic data type that captures both language-agnostic and Java-specific facts in immutable binary relations. These relations represent program information such as existing *declarations*, *method invocations*, *field accesses*, *interface implementations*, *class extensions*, among others [25]. To gather relevant data, Rascal M^3 leverages the Eclipse JDT Core Component[4] to build and traverse the abstract syntax trees of the target Java projects.

We consider the data provided by the *declarations* and *methodInvocation* relations of the M^3 model [25]. Both of them contain a set of pairs $\langle v_1, v_2 \rangle$, where v_1 and v_2 are values representing *locations*. These locations are uniform resource identifiers that represent artifact identities (aka. logical locations) or physical pointers on the file system to the corresponding artifacts (aka. physical locations). The *declarations* relation maps the logical location of an artifact (e.g., a method) to its physical location. The *methodInvocation* relation maps the logical location of a *caller* to that of a *callee*.

Listing 4.1 depicts an excerpt of the M^3 model extracted from the code presented in Fig. 4.1a. The *declarations* relation links the logical location of the method `findBoekrekeningen`, to its corresponding physical location in the file system. The *methodInvocation* relation states that the `getCriteriaBuilder` method of the `EntityManager` type is invoked by the `findBoekrekeningen` method in the current project.

```
m3.declarations = {
<|java+method://StandaardBoekrekeningService/findBoekrekeningen|,
|file:// ... /StandaardBoekrekeningService.java(501,531,<17,4>,<33,5>)|>,
%[...]}
m3.methodInvocation = {
<|java+method://StandaardBoekrekeningService/findBoekrekeningen|,
|java+method://EntityManager/getCriteriaBuilder|>, [...]}
```

Listing 4.1 Excerpt of the M^3 model extracted from Fig. 4.1a.

4.1.2.4 Data Encoder

Once all the method declarations and invocations have been parsed with Rascal, FOCUS represents the relationships among them using a *rating matrix*. Given a project, each row in the matrix corresponds to a declaration, and each column corresponds to an API call. A cell is set to 1 if the declaration in the corresponding row contains the invocation in the

[4] https://www.eclipse.org/jdt/core/

column, otherwise it is set to 0. In Fig. 4.3, we show an example of the rating matrix for an explanatory project p_1 with four declarations $p_1 \ni (d_1, d_2, d_3, d_4)$ and four invocations (i_1, i_2, i_3, i_4). In practice, a matrix is generally big to contain a large number of methods and invocations.

$$\begin{array}{c} \\ d_1 \\ d_2 \\ d_3 \\ d_4 \end{array} \begin{pmatrix} i_1 & i_2 & i_3 & i_4 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Fig. 4.3 Rating matrix for a project with 4 declarations and 4 invocations.

We conceptualized a 3D context-based ratings matrix to model the intrinsic relationships among various projects, declarations, and invocations. The third dimension of this matrix represents a project, which is analogous to the so-called context in context-aware CF systems. For example, Fig. 6.6 depicts three projects $P = (p_a, p_1, p_2)$ represented by three slices with four method declarations and four method invocations. Project p_1 has already been introduced in Fig. 4.3 and, for the sake of readability, the column and row labels are omitted from all the slices in Fig. 6.6. There, p_a is the *active project* and it has an *active declaration* d_a. *Active* here means the artifact (project or declaration), being considered or developed. Both p_1 and p_2 are complete projects similar to the active project p_a. The former projects, i.e., p_1 and p_2 are also called *background data* since they are already available and serve as a base for the recommendation process. In practice, the more background projects we have, the better is the chance that we recommend relevant API invocations.

4.1.2.5 Similarity Calculator

By exploiting the context-aware CF technique, the presence of additional invocations is deduced from similar declarations and projects. Given an active declaration in an active project, it is essential to find the subset of the most similar projects, and then the most similar declarations in that set of projects. To compute similarities, we devised a weighted directed graph that models the relationships among projects and invocations. Each node in the graph represents either a project or an invocation. If project p contains invocation i, then there is a directed edge from p to i. The weight of an edge $p \rightarrow i$ represents the number of times a project p performs the invocation i. Fig. 6.7 depicts the graph for the set of projects in Fig. 6.6. For instance, p_a has four declarations and all of them invoke i_4. As a result, the edge $p_a \rightarrow i_4$ has a weight of 4. In the graph, a question mark represents missing information.

For the active declaration in p_a, it is not known yet whether invocations i_1 and i_2 should be included.

Considering $(i_1, i_2, .., i_l)$ as a set of neighbor nodes of p, the feature set of p is the vector $\overrightarrow{\phi} = (\phi_1, \phi_2, .., \phi_l)$, with ϕ_k being the weight of node i_k. Each constituent weight is computed as the *term-frequency inverse document frequency* value, i.e., $\phi_k = f_{i_k} * log(\frac{|P|}{a_{i_k}})$, where f_{i_k} is the weight of the edge $p \rightarrow i_k$; $|P|$ is the number of all considered projects; and a_{i_k} is the number of projects connected to i_k. Eventually, the similarity between p and q is computed as the cosine between their corresponding feature vectors $\overrightarrow{\phi} = \{\phi_k\}_{k=1,..,l}$ and $\overrightarrow{\omega} = \{\omega_j\}_{j=1,..,m}$, given below:

$$sim_\alpha(p,q) = \frac{\sum_{t=1}^{n} \phi_t \times \omega_t}{\sqrt{\sum_{t=1}^{n} (\phi_t)^2} \times \sqrt{\sum_{t=1}^{n} (\omega_t)^2}} \tag{4.1}$$

Given that $\mathbb{F}(d)$ and $\mathbb{F}(e)$ are the sets of invocations for declarations d and e, respectively, then the similarities between d and e are calculated using the Jaccard similarity index as follows:

$$sim_\beta(d,e) = \frac{|\mathbb{F}(d) \bigcap \mathbb{F}(e)|}{|\mathbb{F}(d) \bigcup \mathbb{F}(e)|} \tag{4.2}$$

4.1.2.6 API Generator

This component is a part of *Recommendation Engine*, and it is used to generate a ranked list of API function calls. As shown in Fig. 6.6, the active project p_a already includes three declarations, and at the time of consideration, the developer is working on the fourth declaration, corresponding to the last row of the matrix. p_a has only two invocations, represented in the last two columns of the matrix, i.e., cells marked with 1. The first two cells are filled with a question mark (?), implying that it is not clear if these two invocations should also be integrated into p_a. *API Generator* predicts additional invocations for the active declaration by computing the missing ratings exploiting the following collaborative-filtering formula [49]:

$$r_{d,i,p} = \overline{r}_d + \frac{\sum_{e \in topsim(d)} (R_{e,i,p} - \overline{r}_e) \cdot sim_\beta(d,e)}{\sum_{e \in topsim(d)} sim_\beta(d,e)} \tag{4.3}$$

Equation 6.4 is used to compute a score for the cell representing method invocation i, declaration d of project p, where $topsim(d)$ is the set of top similar declarations of d; $sim_\beta(d,e)$ is the similarity between d and a declaration e, computed using Eq. (6.3); $\overline{r}_d$ and $\overline{r}_e$ are the mean ratings of d and e, respectively; and $R_{e,i,p}$ is the combined rating of d for i in all the similar projects, computed as follows [49]:

$$R_{e,i,p} = \frac{\sum_{q \in topsim(p)} r_{e,i,q} \cdot sim_\alpha(p,q)}{\sum_{q \in topsim(p)} sim_\alpha(p,q)} \tag{4.4}$$

where $topsim(p)$ is the set of top similar projects of p, k=$|topsim(p)|$ is the number of neighbor projects; and $sim_{\alpha}(p,q)$ is the similarity between p and a project q, computed using Eq. 6.2. Equation 6.5 implies that a higher weight is given to projects with higher similarity. In practice, it is reasonable since, given a project, its similar projects contain more relevant API calls than less similar projects. Using Eq. 6.4 we compute all the missing ratings in the active declaration and get a ranked list of invocations with scores in descending order, which is then suggested to the developer. In Eq. 6.5, a set of k projects is used to compute the ranking, and no matter how large k is, eventually we obtain a real score for each API. Therefore, the final list always contains N items, regardless of k.

In the proposed implementation, we employed a sparse matrix to store the 3D tensor. This allows us to optimize both the storage and computation, and thus increasing the number of neighbor projects for the recommendation. By the current version, FOCUS is able to efficiently compute the recommendations, and maintain a trade-off between computational complexity and effectiveness.

4.1.2.7 Code Builder

This sub-component is responsible for recommending real code snippets to developers. From the ranked list, *top-N* invocations are selected as query to search the corpus for relevant declarations. To limit the search scope, we consider only the most similar projects.

Using the Jaccard index as the similarity metric, for each query, we search for declarations that contain as many invocations of the query as possible. Once the corresponding declarations are identified, their source code is retrieved using the *declarations* relation of the Rascal M^3 model. Thanks to its modularity, Rascal is able to decompile and analyze projects written in different programming languages, e.g., Java [25], C/C++ [8], PHP [108]. Rascal also allows us to compute M^3 model from both source code folders and binaries, e.g., JAR files independently. Thus we implemented a dedicated function that extracts the real source code of a method declaration by means of the computed M^3 model and the project location. Finally, the resulting code snippet is suggested to the developer.

4.1.2.8 Code Recommendation

In Fig. 4.1a, given that *findBoekrekeningen* is the active declaration, the invocations it contains are used together with the other declarations in the current project as the query to feed the recommendation engine. The produced outcome is a ranked list of real code snippets, and we show the top one, named *findByIdentifier*, in Listing 4.2.

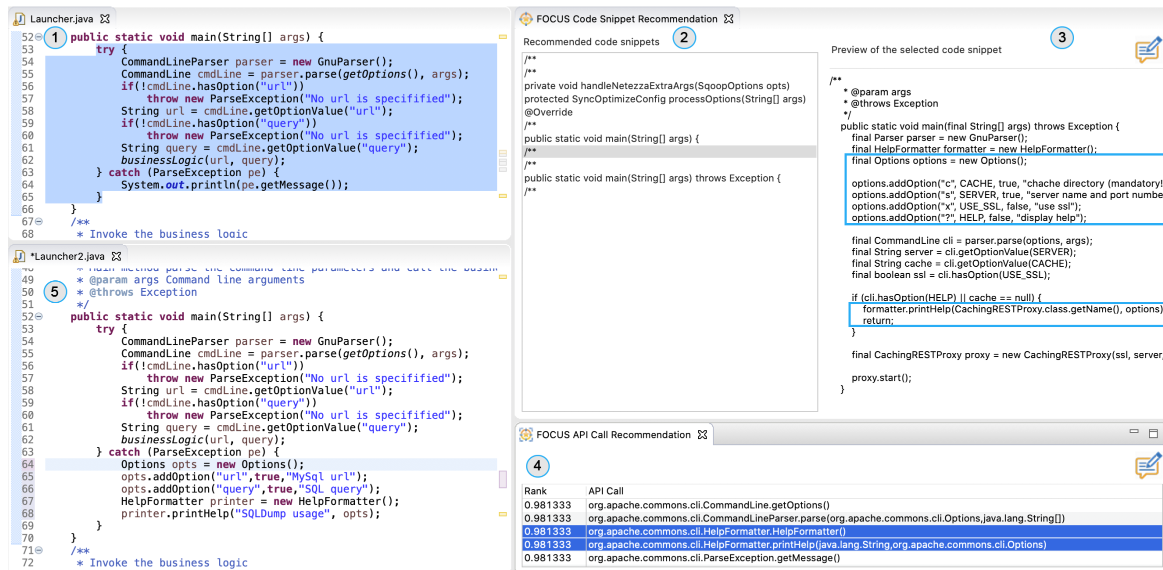


Fig. 4.4 FOCUS IDE.

```
public  List<QuestionsStaged>  findByIdentifier  ( String   identifier  ) {
    log . fine (" getting  Session  instance  by  identifier  :  " +  identifier  );
    try  {
        CriteriaBuilder   cb = entityManager.  getCriteriaBuilder  ();
        CriteriaQuery <QuestionsStaged>  criteria   = cb.createQuery(QuestionsStaged. class );
        Root<QuestionsStaged> qs =   criteria  .from(QuestionsStaged. class );
        criteria  . select (qs) . where(cb.equal (qs. get("  identifier  "),    identifier  ));
        log . fine ("get   identifier   successful ");
        return  entityManager. createQuery(  criteria  ).  getResultList ();

    } catch  (RuntimeException re)  {
        log . severe ("get   identifier   failed " + re);
        throw  re ;
    }
}
```

Listing 4.2 Recommended source code for the snippet in Fig. 4.1a.

By comparing the recommended code and the original one in Fig. 4.1b, we realize that though they are not the same, they indeed share several method calls and a shared intent: both snippets exploit a *CriteriaBuilder* object to build, perform a query, and eventually retrieve some results. Furthermore, the outcome of both declarations is of the *List* type. More importantly, compared to the original code in Fig. 4.1b, the recommended snippet appears to be of a higher quality and robustness.

We conclude that for the motivating example, FOCUS is helpful since the recommended code together with the corresponding list of function calls, i.e., *get*, *equal*, *where*, *select*,

ifnextchar.etcetc., provides the developer with practical instructions on how to use the API at hand to implement the desired functionality.

4.1.2.9 Using FOCUS in the Eclipse IDE

As shown in Fig. 4.4, FOCUS has been integrated into the Eclipse IDE.[5] The figure depicts a real development scenario where a developer is implementing the *SQLDump* project[6] by improving the existing code with recommendations provided by FOCUS. *SQLDump* is a simple command-line utility that exploits the *apache-cli* library[7] to execute an SQL query and export results as a CSV file.

The first implementation of the *main* method prints parameter errors to the console by using Java I/O facilities, i.e., *System.out.println* ①. FOCUS suggests to the developer both code snippets ② and ③, and a ranked list of predicted APIs ④ that are relevant to the code being developed. Furthermore, it recommends a possible improvement that includes the usage of the *HelperFormatter* class ⑤: the *catch* statement block is completely defined and the *System.out.println* invocation is replaced by *HelperFormatter* provided by *apache-cli*. Meanwhile *printHelp* is a method of *HelperFormatter* that prints both possible parameter errors as well as an introduction on how to run *SQLDump* from command line. As a result, with the help of FOCUS, the developer can learn how to use the method both from the code snippets ③ and the list of API calls ④.

The *goal* of this study is to evaluate FOCUS and compare it with two state-of-the-art tools, i.e., UP-Miner [276] and PAM [83], with the *purpose* of determining the extent to which it can provide a developer with accurate and useful recommendations, featuring code snippets containing API usage patterns relevant for the developers' context. The *quality focus* relates to the API recommendation accuracy and completeness, the time required to provide a recommendation, and the extent to which developers perceive the recommendation useful.

PAM has been chosen as baseline for comparison, since it is among the state-of-the-art tools in API recommendation: it has been shown [83] to outperform other similar tools such as MAPO [298] and UP-Miner [276]. To conduct the comparison with PAM, we exploited its original source code which has been made available online by its authors.[8] Furthermore, to facilitate future replications, we published all the artifacts together with the tools used in the evaluation

[5]An instruction on how to install the IDE is available at: https://mdegroup.github.io/FOCUS-Appendix/install.html

[6]https://github.com/aparsons/SQLDump

[7]http://commons.apache.org/proper/commons-cli/

[8]https://github.com/mast-group/api-mining

After formulating the research questions in Section 4.1.3, the following subsections describe datasets, analysis methodology, and the evaluation metrics used to evaluate FOCUS.

4.1.3 Evaluation

4.1.3.1 Research Questions

The conducted study aims to address the following research questions:

▷ **RQ**$_1$: *How does FOCUS compare with UP-Miner and PAM?* Both UP-Miner [276] and PAM [83] are well-founded API recommendation tools. UP-Miner has been shown to outperform MAPO [298], while PAM gains a superior performance compared to both UP-Miner and MAPO. In the previous version of the tool [181], the results demonstrates that FOCUS outperforms PAM on different datasets collected from GitHub and MVN. In this work, we compare FOCUS with UP-Miner and PAM on an Android dataset to further study their performance on a new application domain.

▷ **RQ**$_2$: *How successful is FOCUS at providing recommendations at different stages of a development process?* For a recommender system, it is essential to be able to return relevant recommendations, indicating by a high number of true positives as well as a low number of both false positives and false negatives. This research question evaluates to which extent the tool can provide accurate and complete results.

▷ **RQ**$_3$: *Is there a significant correlation between the cardinality of a category and accuracy?* We examine whether given a testing app, having more apps of the same category is beneficial to the recommendation outcome.

▷ **RQ**$_4$: *Can FOCUS recommend relevant code snippets?* We study if the recommended code snippets provided by FOCUS are relevant to support developers in fulfilling their tasks.

▷ **RQ**$_5$: *How are FOCUS recommendations perceived by software engineers during a development task?* Finally, we are interested in investigating whether FOCUS is useful from a developer point of view. To this end, we conducted a user study to evaluate the relevance of API calls and code snippets provided by FOCUS to support a particular development context. A group of 16 Master's students in Computer Engineering has been involved to assess two real-world development scenarios.

4.1.3.2 First Evaluation: Simulating Developers' Behavior

In the following, we describe the dataset used to address RQ$_1$-RQ$_4$, as well as the data extraction method. As it is explained in Section 4.1.3, for RQ$_5$ we rely on different datasets,

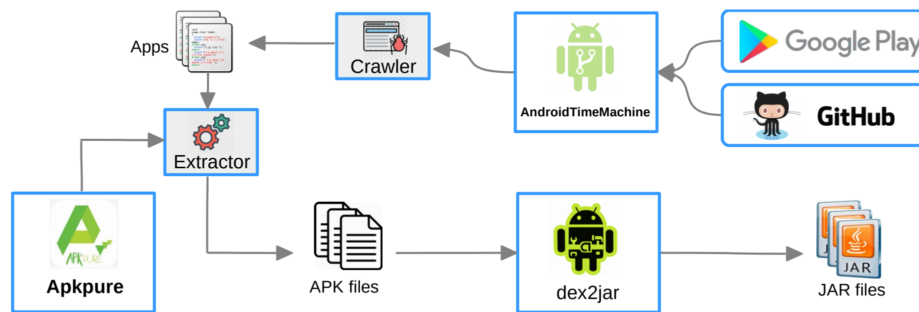


Fig. 4.5 The data extraction process.

because the aim is to let developers leverage FOCUS recommendations, and tasks should be simple enough for an experimental setting.

Evaluation Dataset and Data Extraction - While FOCUS is able to work with different data sources as well as programs written in various languages, the evaluation *context* focuses on the applicability to a specific domain, i.e., Android programming. Although Android development is per se not very different from the development of other kinds of applications, after the evaluation reported in the previous paper featuring heterogeneous Java programs [181], the aim of this evaluation is to show how, by learning from a training set belonging to applications from the same ecosystem, FOCUS is capable of providing accurate recommendations. We have chosen Android not only because of the large availability of data needed to perform an empirical evaluation, but also because recommending API calls and usage patterns is deemed to be important in Android programming [81].

Since FOCUS accepts as input data extracted by Rascal, which in turn requires a specific format, we devised our own method to acquire an Android dataset eligible for the evaluation. The extraction process needs to comply with some certain requirements, and it is illustrated in Fig. 4.5. First, we exploited the *AndroidTimeMachine* platform [91] to crawl open source projects. The platform fetches apps from the Google Play store[9] and associates them with the open source counterparts hosted in GitHub. The crawling process resulted in a set of 7,968 open source Android apps. Most of the apps (82%) in the dataset are written in Java; 4% in Kotlin; 4% in JavaScript, 2% in C++, and 1% in C#. The remaining 7% belong to other languages.

As Rascal can parse certain programming languages, from the initial dataset we filtered out irrelevant projects to select only the Java and Kotlin ones, which account for the majority of the apps. Afterwards, we retrieved the corresponding compiled APK files by querying the Apkpure platform[10] using some tailored Python scripts [237]. The process culminated in

[9] https://play.google.com/
[10] https://apkpure.com/

the final corpus consisting of 2,600 APK binary files (mined from Apkpure) together with additional metadata (mined from Google Play), including authors, categories, star rating, price, and the number of downloads. By carefully inspecting the data, we realized that most of the apps are highly rated and they have a high number of downloads.

We decompiled the APKs into the JAR format by means of the **dex2jar** tool [1]. The JAR files were then fed as input for Rascal to convert them into the M^3 format, which can eventually be consumed by FOCUS.

In total, there are 26,854 API functions in the whole dataset, and most of them are invoked by a small number of declarations (and thus projects[11]): 15,731 APIs are called in only one project. Only a tiny fraction of the APIs is extremely popular by being included in a large number of projects: ten APIs are called in more than 1,900 projects and 15,000 declarations. The most popular API call is *java/lang/StringBuilder/append(java.lang.String)* and it appears in 2,512 projects and 54,828 declarations.

Altogether, this reflects *the long tail effect* which has already been encountered by third-party libraries recommendation [182]. Such an effect can be expressed as follows: For many outcomes, about 80% of consequences originate from 20% of the causes [130]. When we apply this to API recommendation, it is interpreted as: "*About 80% of the APIs come from 20% of the apps.*" As it has been shown in various studies [182, 266], providing products in the long tail is beneficial to the final recommendations. In a similar fashion, we suppose that the ability to suggest APIs rarely included by apps, is of particular importance, as this may help discover useful APIs that have been normally obscured from search engines.

A summary of the categories and their corresponding number of items in the considered dataset is also provided. Due to the space limit, we cannot show and discuss all the figures here. Please refer to the online appendix for more details.[12]

With this dataset, we aim at evaluating if the proposed approach is able to support mobile developers in diverse application domains as well as with various levels of apps' maturity, thereby attempting to resemble real-world development scenarios. We use the collected dataset in RQ_1, RQ_2, RQ_3, and RQ_4 to evaluate FOCUS as well as to compare it with the two baselines.

Finally, the following main steps are conducted to create the required metadata, which can then be used to feed FOCUS.

- the corresponding Rascal M^3 model is generated for every project in the dataset;

[11] For the sake of presentation, from now on the two terms "app" and "project" are used interchangeably.
[12] https://mdegroup.github.io/FOCUS-Appendix/

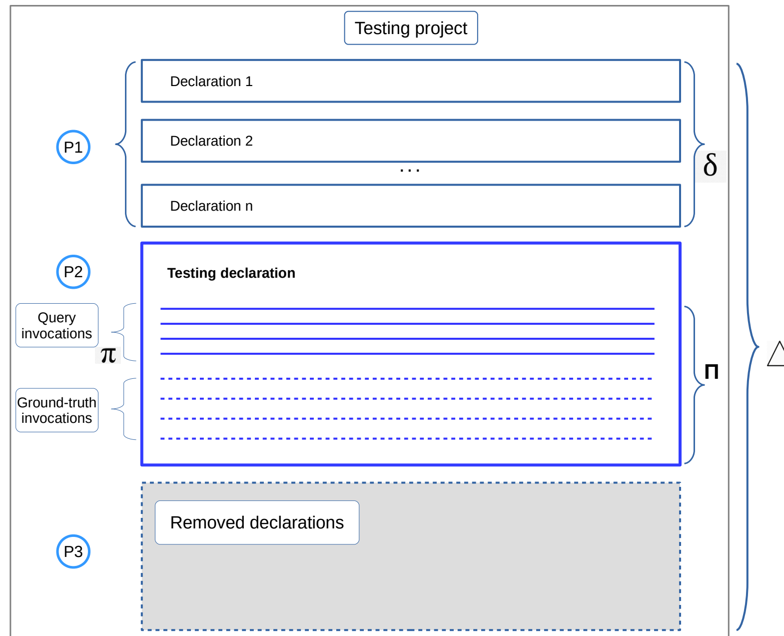


Fig. 4.6 The extraction of data for a testing project.

- the corresponding ARFF representation[13] for each M^3 model is generated in order to be used as input for applying FOCUS and PAM during the actual evaluation steps discussed in the next sections.

How we simulate developers' behavior - To evaluate FOCUS in RQ$_1$-RQ$_4$, we simulate the behavior of a developer who is programming a project and needs practical recommendations to complete it. Figure 4.6 provides an intuition on how the extraction of an active/testing project p_a is done. The project consists of a set of declarations and they are divided into three parts, namely **P1**, **P2**, and **P3**, which are explained as follows.

- **P1**: A set of complete declarations, e.g., Declaration 1, Declaration 2, ifnextchar.etcetc..
- **P2**: A testing declaration, for this declaration, only a portion of code is available to feed the recommendation engine, while the rest is removed and saved as ground-truth data. This corresponds to the scenario in Fig. 4.1a, where the developer is implementing the *active declaration* d_a, and she needs recommendations on the next APIs to be added;
- **P3**: Removed declarations: A certain part consisting of some declarations is removed. This aims to simulate the scenario when the developer is only at an early stage of the project.

[13]https://www.cs.waikato.ac.nz/ml/weka/arff.html

Correspondingly, there are the following parameters:

- Δ is the number declarations in p_a ($\Delta > 0$);
- Only δ declarations ($\delta < \Delta$) are used as input for recommendation and the rest is discarded;
- In total, d_a has Π invocations, however only the first π invocations ($\pi < \Pi$) are selected as testing, and the rest is ground-truth data;
- k is the number of neighbour projects (cf. Section 4.1.2.6);
- Given a ranked list of APIs, the developer typically pays attention to the *top-N* items only, i.e., N is the *cut-off value* for the list.

For d_a, only a half of the code lines of the method's body is selected to feed the recommendation engine. In fact, Rascal can parse only compilable code, thus there might be some compilation errors at some points, where the code is incomplete. As a result, in practice, we suppose that FOCUS can provide recommendations only when the developer temporarily stops at a certain point where the whole declaration becomes compilable. Thus, to increase the applicability of FOCUS, as a developer one should try to make the code compilable as soon as they can by closing open loops, try/catch blocks, return statements, etc. This is supported pretty well by IDEs such as Eclipse which automatically recommend and insert closed loops and try/catch blocks. In this respect, we suppose that in most cases, code is executable, though it is yet complete.

Table 4.1 shows four configurations, i.e., **C1.1**, **C1.2**, **C2.1**, and **C2.2**, corresponding to different combinations of δ and π. Furthermore, **C1.1** and **C1.2** as well as **C2.1** and **C2.2** are pairwise relevant. For example, both **C1.1** and **C1.2** have the same number of method declarations (δ), they differ in the number of invocations in the testing declaration (π).

For the purpose of validation, the original dataset (cf. Section 4.1.3) was split into two independent parts, namely a *training set* and a *testing set*. In practice, the training set represents the OSS projects that have been collected ex-ante, and they are available at the developer's disposal, ready to be exploited for any mining purposes. The testing set represents the project being developed, or *the active project*. We opted for k-fold cross validation [132] as it has been widely chosen to study machine learning models. Depending on the availability of input data, the dataset with n elements is divided into f equal parts, so-called *folds*. For each validation round, one fold is used as testing data and the remaining f-1 folds are used as training data. In our evaluation, two values were selected, i.e., f=10 and f=n. The former corresponds to *ten-fold cross validation* while the latter corresponds

Table 4.1 Experimental configurations.

Conf.	δ	π	Description
C1.1	$\Delta/2-1$	1	Nearly the first half of the declarations is used and the second half is discarded. The last declaration of the first half is selected as the active declaration d_a. For d_a, only the *first* invocation is provided as query, and the rest is used as ground-truth data, i.e., `GT(p)`. This configuration represents an early stage of the development process and, therefore, only limited context data is available to feed the recommendation engine.
C1.2	$\Delta/2-1$	4	Similarly to **C1.1**, almost the first half of the declarations is retained and the second half is removed. d_a is the last declaration of the first half declarations. For d_a, the first *four* invocations are provided as query, and the rest is `GT(p)`.
C2.1	$\Delta-1$	1	The last method declaration is selected as testing, i.e., d_a and all the remaining declarations are used as training data. In d_a, the *first* invocation is kept and all the others are taken out as ground-truth data `GT(p)`. This mimics a scenario where the developer is almost finished implementing p.
C2.2	$\Delta-1$	4	Similar to **C2.1**, d_a is selected as the last method declaration, and all the remaining declarations are used as training data. The only difference with **C2.1** is that in d_a, the first *four* invocations are used as query and all the remaining ones are used as `GT(p)`.

to *leave-one-out cross validation* [287], and they are exploited depending on the purpose as well as the availability of data. With ten-fold cross validation, we shuffle the list of the apps considered in the evaluation, and then randomly split them into ten equal parts. In the evaluation, we attempt to equally distribute the projects into the folds, so as to maintain a balance among the folds with respect to the projects' size. For every experiment, the execution is done ten times: each time one fold is used for testing, and the remaining nine folds are used as training data. Eventually, we averaged out the metrics obtained from the ten folds to get the final results.

To assess the FOCUS's accuracy, we employ a set of metrics defined in Chapter 3.3, i.e., success rate, precision, recall, and Levenshtein distance. Furthermore, we evaluate the time needed for the systems to generate predictions by using a laptop with Intel Core i5-7200U CPU @ 2.50GHz×4, 8GB RAM, and Ubuntu 16.04.

How we address RQ_1-RQ_4 - ▷ **RQ$_1$.** To address RQ$_1$, we compare the performance of FOCUS with that of UP-Miner and PAM. The experience gained in the previous work [181] reveals that PAM cannot scale well with large datasets, i.e., it suffers from a high computational complexity. Meanwhile, FOCUS is more efficient as it is capable of incorporating a large number of background projects and swiftly producing recommendations. In particular, both systems were experimented on a mainstream laptop using a set of 549 training projects with 80MB in size to measure the execution time [181]. On average, PAM requires 320 seconds to provide a recommendation, while FOCUS needs just 1.80 seconds. Through a careful observation on the Android dataset (cf. Section 4.1.3), we realized that many of them are big in size, and a training set of 2,360 apps may add up to more than 2.0GB. This essentially means that it is infeasible to run PAM on the entire dataset, since the execution time may exponentially soar. Thus, for RQ$_1$ we can leverage only a portion of the original corpus. To be more precise, we selected 500 apps of average size. There are 39

categories in total and most of them contain a small number of apps, while *Tools* is still the biggest category with 151 apps, accounting for 30.20% of the total amount. We opted for *leave-one-out cross-validation* [287], aiming to exhaustively exploit the background data. We study the performance of FOCUS by considering all the four configurations listed in Table 4.1, i.e., **C1.1**, **C1.2**, **C2.1**, and **C2.2**. The cut-off value N is used to investigate how accurately the system is able to provide recommendations with respect to different lengths of the ranked list. In RQ_1, we set N to 30, attempting to study the three systems on a long list of recommendations. We also consider, as can be seen in Eq. 6.5, different values of the number of neighbor apps, i.e., k=$\{1,2,3,4\}$. The evaluation was executed 500 times, by each validation, one app is used as testing and all the remaining 499 apps are used for training. To aim for a reliable comparison, we ran UP-Miner and PAM using their original settings in our evaluation.

▷ **RQ_2.** For this research question, we made use of the whole corpus introduced in Section 4.1.3, which contains all the 2,600 collected apps. Moreover, since we have a larger amount of data compared to RQ_1, we employ ten-fold cross-validation in this research question. We analyze the performance of FOCUS for combinations of: *(i)* different configurations, i.e., **C1.1**, **C1.2**, **C2.1**, and **C2.2**; *(ii)* different values of N, i.e., N=$\{1,5,10,15,20\}$; and *(iii)* different values of k, i.e., k=$\{1,2,3,4,6,10\}$. The rationale behind the selection of such specific values is as follows. We should incorporate a certain number of neighbor projects k when computing recommendations, otherwise the matrix will become big (cf. Fig. 6.6), which possibly induces an expensive computational cost. While such a large number of N seems to be unrealistic, in the scope of our evaluation, we have to consider it to ensure the generalizability of our final conclusions. In practice, a small enough number of N items should be presented to the developers, so as to avoid overwhelming them. We report, for different configurations and values of N and k, the success rate, and performance gain. Also, we plot the precision/recall curves for different configurations and values of k.

▷ **RQ_3.** To address RQ_3, we perform controlled experiments on the whole dataset described in Section 4.1.3. Similar to RQ_2, we conducted the experiments following the ten-fold cross-validation methodology. The apps collected in the corpus span over a total of 47 categories, such as *Productivity*, *Communication*, *Music & Audio*, or *Business*. The cardinality (i.e., the number of apps within a category) of the categories varies considerably: most of them contain a small number of apps, i.e., ranging from 1 to 20 items for almost half of the topics. The biggest category with 659 apps is *Tools*, while there are three categories with only two apps, i.e., *Trivia*, *Music*, and *Parenting*.

With this research question, we aim at examining if there is a strong positive correlation between two variables, i.e., the cardinality of a category and the corresponding precision.

In other words, we hypothesize that apps belonging to populous categories might possibly get a better recommendation since they have more, presumably, *relevant* background data, i.e., projects coming from the same domains. This would have an impact in practice as follows: once the developer specifies one or more domains for her app, we can search for recommendations just by looking for apps within the same domains, aiming to narrow down the search scope. This is useful since it contributes to a reduction in the overall execution time. However, this is a pure assumption, which needs to be carefully studied through concrete experiments.

For each category, we computed the precision for all of its constituent apps following Eq. 3.2, and the precision of a category was averaged out over the apps. Eventually, the correlation between the cardinality and precision is computed using the Spearman's rank correlation coefficient and Kendall, i.e., ρ and τ, respectively. The coefficients range from -1 (perfect negative correlation) to +1 (perfect positive correlation), while ρ=0 or τ=0 implies that the variables are not correlated at all. The reason why we compute both Spearman's and Kendall's correlation is because the number of categories is relatively small, and the Spearman's correlation may be more suitable in this case. We do not use the Pearson's correlation as we cannot assume the presence of a linear relationship between categories and precision.

▷ **RQ$_4$.** In this research question, we study if FOCUS is able to recommend source code relevant to the method declaration under development, exploiting the ten-fold cross-validation technique. As an example, we assume that the developer is working on the incomplete code snippet depicted in Fig. 4.1a, and FOCUS is expected to suggest real code such as the one in Fig. 4.1b, or the one in Listing 4.2.

To evaluate the similarity between two declarations, we compare their constituent APIs. This comparison is based on the observation coming from an existing work [158] that if projects or declarations share API calls implementing the same requirements, then they are considered to be more similar than those that do not have similar API usage. Following the same line of reasoning, we evaluate the similarity/relevance between two snippets by examining if they share common API function calls and have the same sequence of these calls.

To address this research question, we leverage the dataset of 500 apps also used to address RQ$_1$. We deliberately make use of such a small dataset due to the following reason: with this dataset, we analyze the ability of FOCUS to recommend relevant code snippets, given that there is a fairly small amount of training data. We conjecture that, as confirmed later in the chapter, if FOCUS works effectively on a small dataset, it will perform well on bigger ones. To evaluate if a recommended snippet is relevant to the query, we measure the level of

similarity between them using the Levenshtein edit distance [141], which has been used by prior work for similar purposes, e.g., tracking source code clones [258]. Given the source code of a declaration d_1, we parse it using Rascal to get the API invocations. Afterwards, we encode each of the invocations using a unique character, resulting in a string s_1. Thus, the evaluation of the similarity between two declarations d_1 and d_2 boils down to comparing the corresponding strings s_1 and s_2, by counting the number of replacements needed to convert s_1 to s_2 using Eq. 3.9. Such a metric takes into account not only the common characters between s_1 and s_2, but also the order in which they appear. Correspondingly, this means that two code snippets are similar/relevant if they share common API function calls as well as have the same sequence of the calls. In this sense, the smaller the distance we get, the more similar the two snippets are, and vice versa.

To simplify the comparison performed in RQ_4, we only used Configuration **C1.2** (cf. Table 4.1). The rationale behind the selection of the configuration is as follows: it represents a more authentic development scenario, corresponding to the situation where the developer already finishes a part of the declaration, and she expects to get recommendations. To be more concrete, given a testing project, we kept the first half of the declarations and removed the second half; the last declaration of the first half declarations is selected as the testing one d_a. For d_a, the first four invocations are provided as query, and the rest is `GT(p)`. Using the *Code Builder* subcomponent (cf. Section 4.1.2), we extracted the real source code of a declaration by means of the computed M^3 model and the project location.

In fact, APK files do not contain source code, thus it is not possible to directly mine real code snippets from the apps. However, FOCUS allows us to extract the method canonical name of a recommended code snippet within the project scope. Moreover, since the dataset is extracted from *AndroidTimeMachine*, there is a mapping between open-source Google store apps with their corresponding repositories. To locate the right pair of APK file and GitHub repository, we check the snapshot date when the mapping was created. In this way, we are able to trace back to the original source code for those apps that have a counterpart in GitHub. Eventually, FOCUS is able to recommend source code, as long as the corresponding app is associated with a source project rooted in GitHub.

4.1.3.3 Second Evaluation: User Study

In this section, we study FOCUS's usefulness of code and API recommendations by means of a task-based user study to address RQ_5. The *goal* of this study is to evaluate FOCUS, with the purpose of understanding whether it could help developers with their implementation tasks. The *quality target* of the study is the perceived usefulness that developers have of recommendations (code snippets and APIs) provided by FOCUS. Therefore, the baselines

Table 4.2 Task assignments to the evaluator groups.

	T1	T2
Group I	Apache-cli, using FOCUS	gson
Group II	Apache-cli	gson, using FOCUS
Group III	gson	Apache-cli, using FOCUS
Group IV	gson, using FOCUS	Apache-cli

presented in the previous section have been not considered since the aim is to assess the qualitative aspects of FOCUS. The *context* consists of participants, i.e., 16 Master's students in Computer Engineering, and objects, i.e., programs involving command line argument parsing and HTML download/parsing.

Study Design and Tasks - As shown in Table 4.2, the experimental design is a crossover design in which participants were split into four groups (each participant worked individually, but each group received the same treatment). Each participant had to carry out two implementation exercises, one using FOCUS recommendations and another without the availability of FOCUS. Different groups featured different ordering of the treatments, to mitigate any ordering/learning effect.

```
/**
* Create apache-cli options for the following elements:
*  url (Mandatory),
*  username (Mandatory): -user <user>
*  password (Mandatory): -pass <password>
*  query (Mandatory): -sql <query>
*  CSV file path: -f - file < filepath >
*  includeHeaders: -headers
*  All the options contains and argument, with the exception of includeHeaders
* @return Available command line options
*/
public static Options getOptions() {
        final Options options = new Options();
        final Option urlOption = new Option("url", true, "database url <jdbc:subprotocol:subname>");
        final OptionGroup urlGroup = new OptionGroup();
        urlGroup.setRequired(true);
        urlGroup.addOption(urlOption);
        options.addOptionGroup(urlGroup);

        //COMPLETE THE METHOD
        return options;
}
```

Listing 4.3 The commons-cli *getOption()* partially implemented method.

The two tasks focus on the usage of different libraries, i.e., *commons-cli*[14] and *jsoup*,[15] and require the completion of three partially implemented methods. *commons-cli* provides

[14] https://commons.apache.org/proper/commons-cli/
[15] https://jsoup.org/

APIs for parsing command-line options passed to programs, while *jsoup* is a library for parsing and manipulating HTML pages using the best of DOM, CSS, and jquery-like methods.

```
@Test
public void getOptionTest () throws IOException {
        Options options = Launcher.getOptions ();
        assertEquals (6, options . getOptions () . size ());
}

@Test
public void parseOKTest() throws Exception {
        String [] arguments = new String []{ "-url", "a",
                "-pass", "pass",
                "-user", "user",
                "-sql", "sql"};
        assertEquals (4, Launcher.parse (arguments). size ());
}

@Test
public void printUsageTest () throws IOException {
        assertNotEquals ("", Launcher.printUsage ());
}
```

Listing 4.4 The unit tests for checking the correctness of the task.

For the tasks with *commons-cli*, the participants completed three methods by: *(i)* implementing a method for specifying the command-line options (we provided the evaluator with the parameter list); *(ii)* parsing the command line parameters and throwing an exception if the mandatory ones are missing; *(iii)* handling parsing exception by printing possible options to the console. Listing 4.3 shows an example of the partial implementation and the method requirements for specifying options. For a detailed description of the two performed tasks, due to space limit, interested readers are kindly referred to our online appendix.[16]

For each method to be completed, we provided (for treatments having the availability of FOCUS) each evaluator with the *top-5* snippets and *top-20* method invocations recommended by FOCUS by giving the initial and partial method implementation as input.

Study Operation - Under the circumstance in which the experiment was conducted, it was neither possible to perform the experiment in a laboratory[17] nor to ask participants to return the results immediately. Instead, each participant could perform the tasks offline and return them to us. Before the study, we performed a laboratory introductory session in video conference, in which we introduced to participants the laboratory goals and tasks (without details about our research question, to avoid biasing them), and left them a detailed instruction documents.

[16] https://mdegroup.github.io/FOCUS-Appendix/tasks.html
[17] Due to the COVID-19 emergency in 2020

During the tasks, participants could access any resource available on the Internet, besides FOCUS recommendations when available based on the study design. Once a participant finished the tasks, s/he had to complete a questionnaire[18] consisting of the following questions: *(i)* three general questions asking about their experience in programming and code search engine; *(ii)* four questions, in a 5-level Likert scale [196], related to the understandability and complexity of the assigned tasks; and *(iii)* four questions to evaluate the relevance and usefulness of the recommendations provided by FOCUS.

Moreover, we asked the participants to submit their implementations. Such implementations have been used for understanding the correctness of the resulting code. For each method to be completed, we defined a specific JUnit[19] unit test for checking their correctness. We did not provide the evaluator with the test methods to avoid bias towards the experiment. Listing 4.4 reports the simple testing methods used to check the correctness of the submitted task. Although the unit tests are rather simple, they have been able to effectively catch any possible implementation failures.

Then, we involved a senior developer experienced with Java programming, *gsoup* and *commons-CLI* libraries to further investigate the method implementations where the unit test fails. The senior developer checked the severity of the identified errors and discarded those that are not related to the usage of the involved library. For instance, some evaluators named the parameters differently, e.g., they used *password* instead of *pass* or *username* instead of *user*. Consequently, the dedicated *parseOKTest* test fails because of a wrong parameter naming. We marked this type of failure as a minor one, and we considered the implementation as correct for the evaluation scope.

How we address RQ_5 - There are the following analyses to address $\mathbf{RQ}_5$:

- We perform a Wilcoxon signed rank test [285] to determine whether there is any statistically significant difference between the number of passed tests for tasks implemented with and without FOCUS (H_0: *there is no significant difference between the percentage of tests passed with and without the availability of FOCUS*). Also, we compute the Cliff's delta effect size [99].

- As for the questionnaire results, we report them using diverging stacked bar charts and discuss them.

[18] https://forms.gle/uoqSTaQ94PArdUST6
[19] http://junit.org/junit4

4.1.4 Results

This section analyzes the experimental results obtained through the evaluation by referring to the four research questions mentioned in Section 4.1.3.

Table 4.3 Success rate of PAM and FOCUS.

	UP-Miner	**PAM**	**FOCUS**			
	—	—	k=1	k=2	k=3	k=4
C1.1	41.66	49.60	81.20	81.60	82.00	**83.80**
C1.2	37.33	52.20	89.81	91.10	92.80	**93.22**
C2.1	44.10	58.22	77.00	78.20	78.60	**79.60**
C2.2	40.66	58.40	89.60	90.20	91.60	**92.10**

RQ$_1$: *How does FOCUS compare with UP-Miner and PAM?* Table 4.3 reports the success rate for PAM and FOCUS, considering different configurations and values of k representing the number of neighbor apps. The cut-off value N was set to 30, attempting to investigate the systems' performance for a long list of recommendations. The table shows an evident outcome: FOCUS always achieves a much better success rate than that of PAM and UP-Miner by all the configurations. For instance, with **C1.2**, FOCUS gets 89.81%, 91.10%, 92.80%, and 93.22% as success rate by k=1, k=2, k=3, and k=4, respectively, while PAM and UP-Miner get 52.20% and 37.33%, respectively. With **C2.2**, FOCUS gets a maximum success rate of 92.10%, which is superior than 58.40% and 40.66% obtained by PAM and UP-Miner, respectively. We further confirm the claim by Fowkes and Sutton [83], i.e., PAM outperforms UP-Miner also in our setting. Concerning the execution time, FOCUS spends 8×10^{-3} seconds to produce recommendation for one app, while UP-Miner and PAM need 3.8×10^{-4} and 1.6 seconds, respectively to perform the same task. In other words, on the given dataset, UP-Miner is the most efficient tool in terms of timing, while FOCUS is much faster than PAM.

The performance gain obtained by FOCUS is understandable in the light of the following arguments. UP-Miner works on the basis of clustering techniques and it is dependent on the similarity among groups of APIs. In other words, UP-Miner computes similarity at the sequence level, i.e., invocations that are usually found together. PAM is a complex system, which consists of six building blocks, i.e., probabilistic model, inference, learning, inferring new patterns, candidate generation, and mining interesting patterns. The system uses a probability distribution to define a distribution over all possible API patterns present in client code, based on a set of API patterns. It also employs a generative model to infer the most probable patterns from ARFF files. Finally, the system generates candidate patterns by relying on the highest support first rule, i.e., searching for the best candidate earlier. Due to these technical details, both UP-Miner and PAM can recommend APIs that commonly

Table 4.4 Success rate (%) for $k = \{2,3,4,6,10\}$ and $N = \{1,5,10,15,20\}$.

N	C1.1					C1.2				
	k=2	**k=3**	**k=4**	**k=6**	**k=10**	**k=2**	**k=3**	**k=4**	**k=6**	**k=10**
1	67.46	69.10	71.34	74.00	75.76	85.69	87.53	88.19	89.38	89.96
5	76.84	78.42	80.38	82.53	84.80	91.11	92.42	92.84	93.88	94.53
10	80.46	82.30	83.42	85.38	87.84	92.80	93.92	94.50	94.92	96.03
15	81.76	84.05	84.84	87.15	89.15	93.69	94.61	95.23	95.61	96.50
20	82.80	84.88	86.23	87.88	90.11	94.07	94.96	95.61	96.11	96.92
	C2.1					**C2.2**				
1	66.30	68.11	70.50	72.65	75.42	82.84	85.50	86.92	88.15	88.96
5	77.03	78.15	77.80	79.57	82.03	90.07	91.00	91.84	92.11	93.42
10	79.46	80.57	80.26	81.96	84.57	91.65	92.50	93.19	94.00	95.11
15	80.76	82.07	81.57	83.76	86.23	92.11	93.30	93.80	94.73	96.00
20	81.73	84.00	84.92	87.34	89.07	92.65	93.88	94.26	94.92	96.15

appear in different code snippets. In contrast, FOCUS is able to consider similarity both at the project level and the declaration level. Therefore, given an active project, FOCUS mines API calls from the most similar declarations in the most similar projects. As a result, this allows FOCUS to outperform both UP-Miner and PAM in finding invocations that fit well to a given context.

It is worth noting that FOCUS gets a considerably high performance, given that the dataset is fairly small. The maximum success rate obtained by **C1.2** and **C2.2** is 93.22% and 92.10%, respectively. Compared to the previous work [181], where a set of 200 GitHub projects was considered to compare FOCUS with PAM, we see that FOCUS substantially improves its recommendations when more data is incorporated into the training. A feature of the considered datasets which may affect the results obtained by FOCUS is the level of dependencies in Android apps compared to that of the GitHub projects. In particular, by counting the number of unique APIs in each app/project for both the Android dataset and the GitHub dataset we see that the former contains more APIs compared to the latter. Many apps have more than 400 unique APIs, meanwhile, most of the GitHub projects have less than 200 unique APIs. This is further supported by previous work [273, 222], which gives evidence that Android projects make heavy use of third-party libraries as well as native libraries.

Answer to RQ$_1$. Though UP-Miner is the most efficient tool, FOCUS substantially outperforms both UP-Miner and PAM in terms of prediction performance. Moreover, FOCUS mines better on Android apps with respect to GitHub projects.

RQ$_2$: *How successful is FOCUS at providing recommendations at different stages of a development process?* In this research question, we are interested in understanding the *completeness* and *accuracy* of FOCUS's recommendations at different project's development stages. For the former, we analyze the corresponding success rate and performance gain,

Table 4.5 Performance gain (%) among the configurations.

Gain of C1.2 w.r.t. C1.1					
N	**k=2**	**k=3**	**k=4**	**k=6**	**k=10**
1	27.02	26.67	23.62	20.78	18.74
5	18.57	17.85	15.50	13.75	11.47
10	15.34	14.12	13.28	11.30	9.32
15	14.59	12.56	12.25	9.71	8.24
20	13.61	11.88	10.88	9.37	7.56
Gain of C2.2 w.r.t. C2.1					
1	25.14	25.53	23.29	21.34	17.95
5	16.93	16.44	18.05	15.76	13.89
10	15.34	14.81	16.11	14.69	12.46
15	14.05	13.68	14.99	13.10	11.33
20	13.36	11.76	11.00	8.68	7.95

while for the latter, we take into consideration the obtained *precision* and *recall* values. Furthermore, we investigate the system's ability to recommend APIs in *the long tail*.

▷ **Success rate.** Table 4.4 compares the success rates obtained by the considered experimental settings. For the smallest cut-off value N, i.e., N=1, FOCUS is still able to provide matches. For instance, with **C1.1** when k=2, the system gets 67.46% as success rate, and this score increases linearly along N: FOCUS gets a success rate of 76.84% and 82.80% when N=5 and N=20, respectively. By Configuration **C1.2**, compared to **C1.1**, we see a sharp increase in performance by all the cut-off values. Take as an example, with k=2, we get 91.11% as success rate for N=5, and the score goes up to 94.07% when N=20. This demonstrates that FOCUS is capable of providing good match even when the developer wants to see a fairly short ranked list. Similarly, by **C2.1** and **C2.2**, FOCUS enhances its success rate alongside k and N.

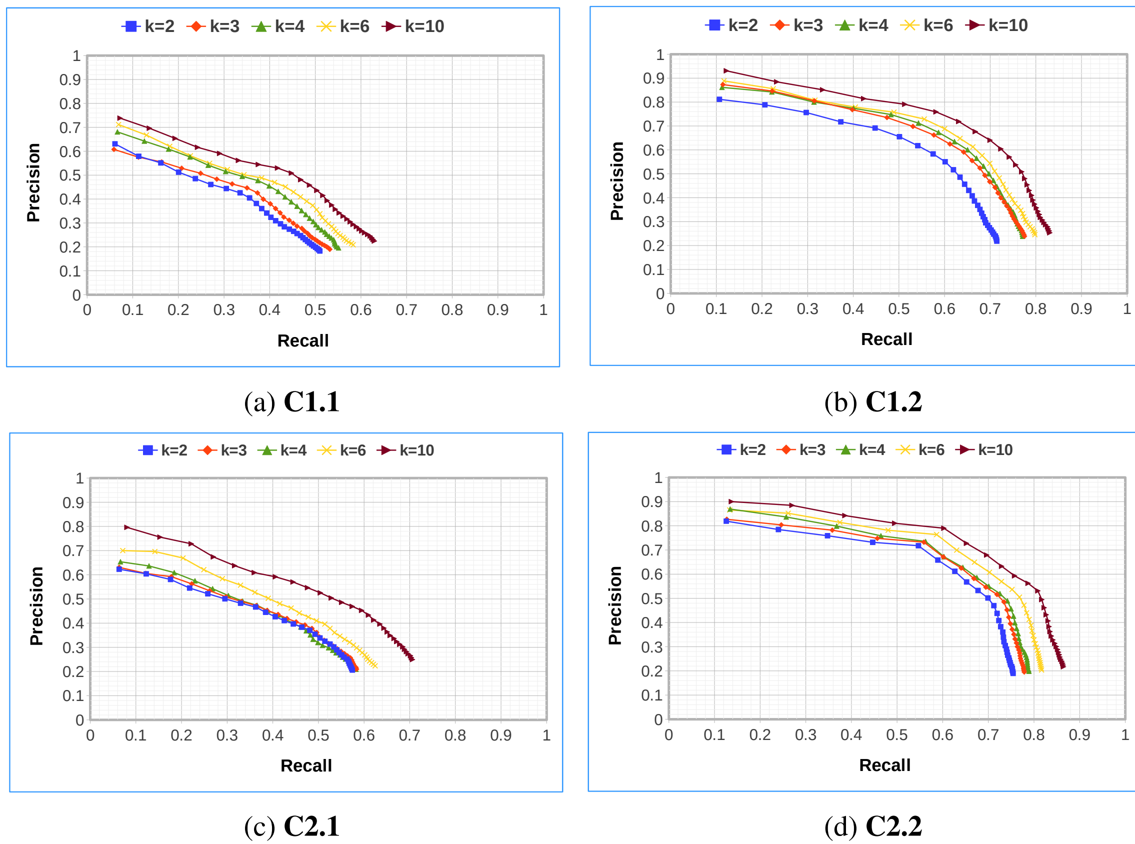
Next, we investigate the effect of changing the number of neighbor apps used in computing recommendations, i.e., k, on the final outcome by comparing the results columnwise. It is evident that when incorporating more neighbors for computing recommendations, FOCUS yields a better success rate. For instance, with **C1.1**, considering the success rate obtained by k=2 and k=3, we see that there is always a gain in performance: for N=5, FOCUS obtains 76.84% and 78.42%, respectively. This score improves substantially when we use more neighbor projects to compute recommendations. Take as an example, FOCUS has a success rate of 71.34% when k=4 and 75.76% when k=10. In summary, FOCUS is more accurate if additional apps are considered for computing the missing ratings in the 3D matrix.

▷ **Performance gain.** Referring to Table 4.1, we see that **C1.1** and **C1.2** as well as **C2.1** and **C2.2** are pairwise comparable. For instance, both **C1.1** and **C1.2** share the same amount of method declarations (δ), they only differ in the number of invocations used in the testing declaration (π). Thus, to investigate the effect of changing π on the recommendations, we consider each pair of related configurations. The results in Table 4.4 indicate a sharp rise

in performance when the configurations change from **C1.1** to **C1.2**. Take as an example, when k=2 and N=1, that means we consider only the first item in the ranked list, FOCUS obtains 85.69% as success rate which is much better than 67.46%, the score yielded by **C1.1**. When k=10 and N=20, the maximum success rate for **C2.1** and **C2.2** is 90.11% and 96.92%, respectively. This suggests that incorporating more invocations, e.g., four instead of one invocation, helps FOCUS significantly enhance its overall performance. In practice, this means that given a declaration, the system is able to provide more accurate recommendations proportionally to the project's maturity.

Given the results in Table 4.4, we analyze the performance gain in percentage (%) and report them it in Table 4.5. The green color and various levels of density are employed to represent the corresponding magnitude. From the table, it is evident that the color gradually fades when we move from left to right, top to bottom, implying that the enhancement goes down linearly when we increase k and N. For example, the correlation between **C1.1** and **C1.2** is as follows: for N=1 the gain is 27.02% with k=2, and it decreases to 26.67% and 23.62% with k=3 and k=4, respectively; when k=10, the gain boils down to 18.74%. The same trend can be seen for other values of k and N. Likewise, the improvement obtained by **C2.2** in comparison to **C2.1** shares a similar pattern: it is big with low k and N, and small with higher k and N. For instance, it reaches 25.14% for N=1 and k=2 and shrinks to 7.95% for N=20 and k=10. Overall, this essentially means that while we get performance gain by incorporating more neighbors, at a certain point, such the gain becomes saturated and there will be no further improvement.

▷ **Accuracy.** We report the accuracy achieved by all configurations using the precision recall curves (PRCs) depicted in Fig. 4.7a, Fig. 4.7b, Fig. 4.7c, and Fig. 4.7d. The cut-off value N has been varied from 1 to 30, aiming to study FOCUS's performance further down in the ranked list. First, we examine the effect of changing k on the precision recall curves. In fact, a system gets a good performance if its precision and recall are high at the same time, and this corresponds to a PRC close to the upper right corner of the diagram. From the figures, it is clear that incorporating more neighbor apps in computing recommendations results in a better accuracy by all configurations. For instance, with **C1.1**, we see a performance gain when increasing the number of neighbor apps: the best precision and recall are 0.75 and 0.63, respectively, and they are obtained when k=10; while by other value of k, i.e., $k = \{2,3,4,6\}$, the system gets a lower precision and recall. Similarly by other configurations, k=10 is also the number of neighbor apps used for computing ratings that contributes to the best accuracy: with **C1.2**, FOCUS achieves 0.92 as precision and 0.84 as recall. With **C2.1** and **C2.2**, the gain in performance when using 10 apps for computing recommendations becomes more evident, in comparison to other values of k, i.e., $k = \{2,3,4,6\}$. This is consistent with the



(a) **C1.1** (b) **C1.2**

(c) **C2.1** (d) **C2.2**

Fig. 4.7 Precision and recall curves obtained for the configurations.

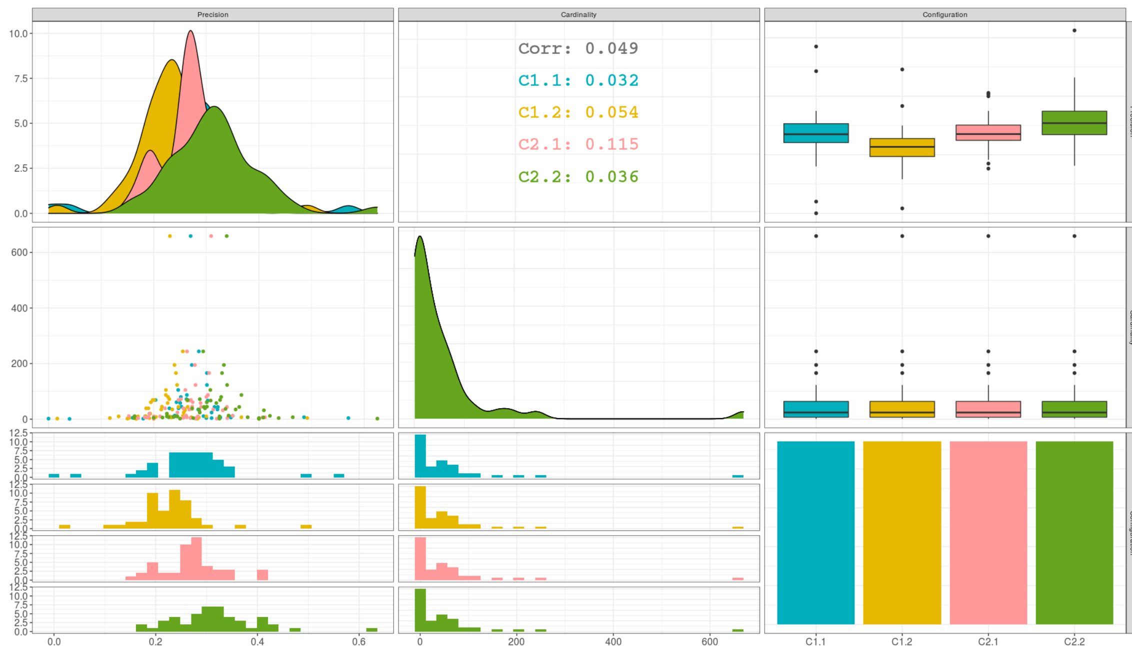


Fig. 4.8 Bivariate analysis of Precision and Cardinality.

outcomes we got by the success rate scores presented in Table 4.4: the system achieves a better performance if it incorporates more similar apps for computing recommendations.

In conclusion, we see that the performance of FOCUS using **C1.2** is superior to that when using **C1.1**. Similarly, compared to **C2.1**, the accuracy obtained by FOCUS using **C2.2** improves substantially, i.e., by equipping the query with more invocations. These facts further confirm that FOCUS is able to recommend more relevant invocations when the developer intensifies the declaration by adding more code. As can be seen in Eq. 6.5, when more invocations are available, the similarity among declarations can be better determined, resulting in a gain in performance.

▷ **The long tail.** We counted the APIs that are recommended more often by FOCUS. By carefully checking the top 20 recommended items, we realized that most of them reside in the long tail. For example, the *java/lang/StringBuilder/toString()* API has been provided 190 times by FOCUS, being the top most recommended item. However, this invocation is only ranked 646 in the list of all the APIs in the dataset. Altogether, this is to show that while recommending very popular APIs may make sense, FOCUS goes far beyond that by recommending also items in the long tail. This is achieved since FOCUS mines API from highly similar projects, given an active project.

Answer to RQ$_2$. FOCUS provides more accurate predictions when more similar projects are incorporated for recommendation. Moreover, it is capable of suggesting APIs in the long tail. Given an active declaration, the system improves its accuracy while the developer keeps coding.

RQ$_3$: *Is there a significant correlation between the cardinality of a category and accuracy?* Table 4.6 depicts the Spearman coefficients for all configurations, with respect to different numbers of cut-off values N. The Kendall coefficients (τ) are comparable to the Spearman ones (ρ), so we omitted them from the table, for the sake of clarity.

By examining the results in Table 4.6 we see that, despite some fluctuations, mainly with **C2.1** and **C2.2**, ρ is considerably small, i.e., the maximum value is ρ=0.160 for **C1.2** and N=25. More importantly, most of the scores are close to 0, indicating an extremely low (e.g., by $N = \{5,20,25\}$ with **C1.1**) or almost no correlation (e.g., by $N = \{15\}$ with **C1.1** and **C2.2**).

Table 4.6 Correlation (ρ) between cardinality and precision, $N = \{5,10,15,20,25\}$.

N	C1.1	C1.2	C2.1	C2.2
5	0.059	0.150	0.068	-0.157
10	0.117	0.132	0.086	-0.013
15	0.001	0.115	0.080	0.005
20	0.046	0.150	0.156	0.054

As an example, Fig. 4.8 depicts precision and cardinality as well as their correlation for N=25. The variables are shown both on the x-axis and y-axis, however at different parts of the axes. This allows us to comprehensively represent the relationship between the two variables for all the four configurations. In particular, on the top-left corner, there is the histogram of precision with respect to cardinality, while the other bar charts at the bottom show the histogram for each of them individually. The middle frame in the top row specifies the correlation coefficients between precision and cardinality for all the configurations. Results show that there is a very weak correlation between the two variables. For instance, the coefficient is 0.032 for **C1.1**, or 0.036 for **C2.2**. As a whole, this unfortunately contradicts our initial conjecture: apps belonging to major categories do not get a better recommendation, although they have in principle, more background data. This means that *searching for recommendations just by looking at apps of the same domain(s) does not guarantee that we will gain benefit.* We attempt to ascertain the possible causes in the following.

According to a previous work [158], if projects share API calls implementing the same requirements, then the projects are considered to be more similar than those that do not have similar API usage. We computed similarity among apps using the *Similarity Calculator* component presented in Section 4.1.2. Such a similarity is measured based on the constituent API function calls of an app (cf. Fig. 6.7 and Eq. 6.2). By carefully examining the final results, we realized that generally, similar apps do not originate from the same domain. To be concrete, considering a ranked list with five items for all 2,600 apps, i.e., N=5, the percentage of items that have similar apps coming from 1, 2, 3, 4, and 5 categories is 1.14%, 6.6%, 21.42%, 41.8%, and 29.0%, respectively.

For instance, **machinekit.appdiscover**[20] belongs to *Libraries & Demo*, however its highly similar apps are from *Education*, *Books & Reference*, *Health & Fitness*, and *Tools*. Since FOCUS relies on the similarity function (cf. Section 4.1.2), it may retrieve invocations from projects in completely different domains to generate recommendations. This explains why projects of a category with a low number of items still get a good accuracy, resulting in a weak correlation between the cardinality of a category and accuracy. In a nutshell, *there exists no correlation because even apps belonging to different categories still contain similar API usage.*

Though the experiment suggests that we cannot save time by looking into some certain categories, on the bright side, it reveals an interesting feature of FOCUS: the tool is able to discover API calls from a wide range of apps, regardless of their origins.

Answer to RQ$_3$. There is no direct correlation between the cardinality of a category and prediction accuracy. Moreover, FOCUS is capable of mining API calls from apps belonging to various application domains.

RQ$_4$: *Can FOCUS recommend relevant code snippets?* As shown in Section 4.1.1, by using the incomplete code in Fig. 4.1a together with other testing declarations as query to feed FOCUS, we obtained a relevant snippet depicted in Listing 4.2, and this is just one of many good matches we got. To provide a concrete analysis, Fig. 4.9 depicts the distribution of the 500 apps dataset with respect to the number of projects (x-Axis) and Levenshtein distance between the testing declaration and the corresponding project (y-Axis).

To facilitate a better view, we mark the apps as four separate clusters. Almost a quarter of the projects or 24% corresponding to 120 projects get zero as the final result, i.e., the distance between the recommended snippet and the original one is zero. This means that for each of these projects, the recommended declaration perfectly matches the original one. By the remaining ones, 23 projects among them accounting for 4.6%, have a distance of one, which also indicates a high level of code similarity. Almost a half of the dataset, i.e., 233 apps corresponding to 46.60%, have a distance being larger than nine.

Figure 4.9 shows that, while FOCUS gains a good recommendation performance for a considerably large number of apps, it *fails* to retrieve matches for some others, i.e., the corresponding Levenshtein distance is large, meaning that the recommended snippets are not relevant to the ground-truth ones. For instance, one project has a distance of 52, or another has a distance of 43. We attempt to find out the rationale behind this outcome. Our main intuition is as follows, by those projects with a large Levenshtein distance, there is a lack

[20] https://bit.ly/3pGKKIL

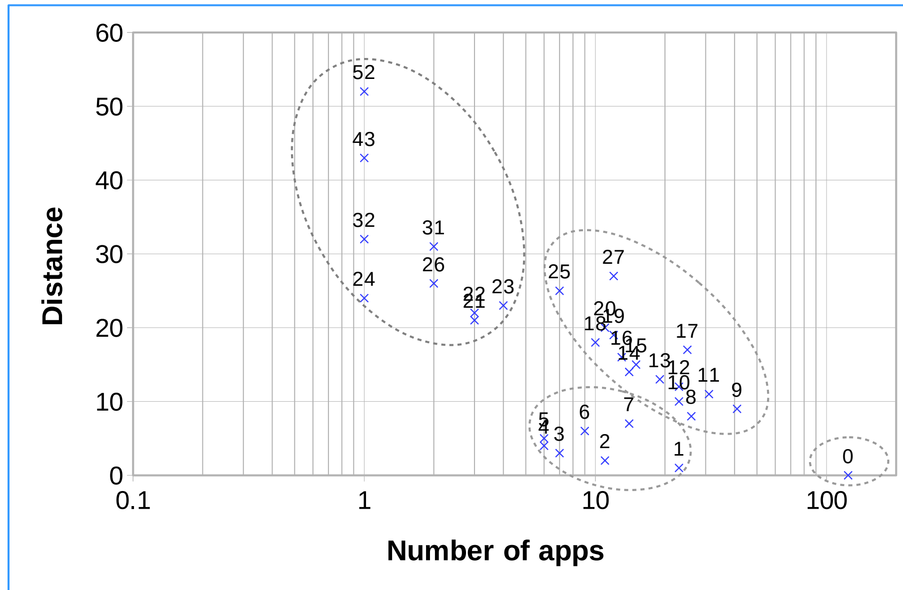
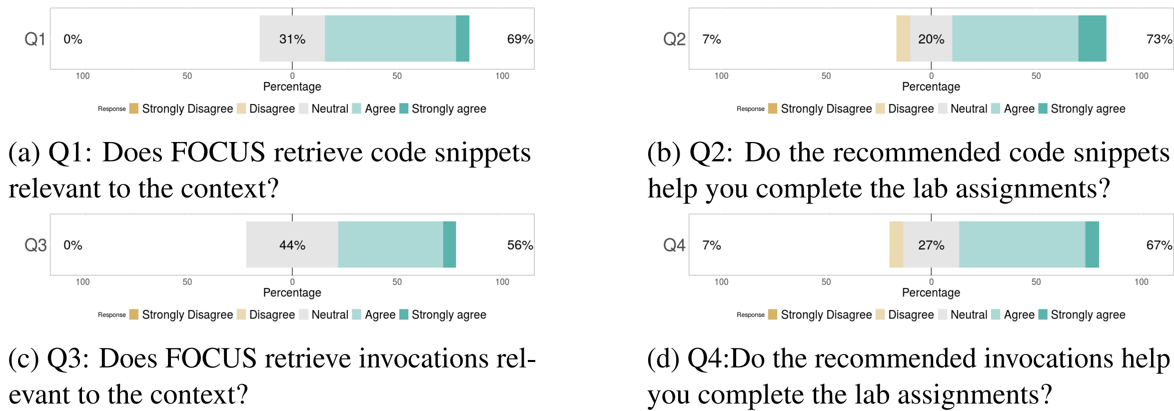


Fig. 4.9 Levenshtein distance for the set of 500 apps.

of relevant training data. In other words, if there are not enough similar projects, FOCUS cannot discover API invocations which eventually fit to the active declaration.

To validate the hypothesis, the following test was conducted: we computed the precision scores for all projects, and compared them with the Levenshtein distances using the Spearman's rank correlation coefficient (Similar to RQ_3). The resulting score is ρ=-0.514, with p-value < 2.2e-16. This can be interpreted as follows: the obtained precision is disproportionate to the Levenshtein distance, or put another way, the higher the precision we get, the shorter the distance, and vice versa. The finding consolidates our assumption: if FOCUS achieves a high precision, it will be able to recommend more relevant code snippets. Furthermore, as we already proved in RQ_3, FOCUS gets a higher precision if we use more similar apps for computing recommendations. Altogether, we conclude that the proposed approach is able to return relevant code snippets if it is fed with more training data. From the set of apps with a Levenshtein distance of 0, we enumerated the APIs and sorted them in descending order to see which invocations have been recommended most. We got a similar outcome of RQ_2: FOCUS recommends several APIs which appear late in the ranked list of the most popular invocations.

Answer to RQ_4. FOCUS can provide relevant source code snippets to a testing declaration, as long as we feed it with a rich training dataset, i.e., there are more projects similar to the one being considered.



(a) Q1: Does FOCUS retrieve code snippets relevant to the context?

(b) Q2: Do the recommended code snippets help you complete the lab assignments?

(c) Q3: Does FOCUS retrieve invocations relevant to the context?

(d) Q4:Do the recommended invocations help you complete the lab assignments?

Fig. 4.10 Results for evaluating the usefulness of the recommendations.

RQ$_5$: *How are FOCUS recommendations perceived by software engineers during a development task?* As explained in Section 4.1.3, 16 participants took part in the user study. Among them, 30% and 50% of them have three years and more than four years of programming experience, respectively. Most of them use a code search engine in a daily basis. Moreover, 80% of the participants agree that the tasks are clear and easy.

First, we analyzed whether the use of FOCUS could help participants produce more correct code. The median number of passed tests was 2 out of 3 both with and without FOCUS. We compared (pairwise, by participant) the percentage of passed test cases using a Wilcoxon signed-rank test. The test did not indicate a statistically significant difference (p-value=0.88), i.e., the correctness of the produced implementations did not change with and without FOCUS. Also the Cliff's d effect size is negligible (d=0.01).

We therefore looked at the perceived usefulness of the recommendations, in terms of API invocations and code snippets. Results of the questionnaire related to task assignments shown in Table 4.2 are shown in Fig. 4.10a, Fig. 4.10b, Fig. 4.10c, and Fig. 4.10d.

Concerning the first question "*Q1: Does FOCUS retrieve code snippets relevant to the context?*" following Fig. 4.10a, 69% of the participants *agree* and *strongly agree* with the fact that the snippets are relevant, while the remaining 31% of them have no concrete judgment on the results, i.e., neutral. This means that most of the developers find that the recommended code snippets fit their programming tasks.

By the second question: "*Q2: Do the recommended code snippets help you complete the lab assignments?*" as shown in Fig. 4.10b, most of the participants find that the snippets recommended by FOCUS are helpful to solve the tasks. In particular, 73% of them agree and strongly agree with the question.

With the third question: "*Q3: Does FOCUS retrieve invocations relevant to the context?*" we are interested in understanding whether FOCUS can fetch invocations related to the given

context. The results in Fig. 4.10c suggest that more than a half of the evaluators, i.e., 56% think that the provided APIs are relevant, while 44% of them have no concrete judgment.

Finally, the results in Fig. 4.10d, corresponding to the last question: "*Q4: Do the recommended invocations help you complete the lab assignments?*" show that 7% of the participants disagree that the APIs are useful, while 27% of them feel neutral about the results. Still, most of them, i.e., 67%, appreciate the recommended APIs, which are helpful to solve their tasks.

Altogether, the Likert scores indicate that FOCUS provides decent recommendations: both the suggested APIs and code snippets are meaningful to the given contexts.

Answer to RQ$_5$. The majority of the study participants positively perceived the context-specific relevance and the usefulness of the recommendations (APIs and code snippets) provided by FOCUS.

4.1.5 Threats to validity

The main threat to *construct validity* concerns the simulated setting used to evaluate the approaches, as opposed to performing a user study. We mitigated this threat by introducing four configurations that simulate different stages of the development process. In a real development setting, however, the order in which one writes statements might not fully reflect our simulation. Also, in a real setting, there may be cases in which a recommender is more useful, and cases (obvious code completion) where it is less useful. This makes a further evaluation involving developers highly desirable.

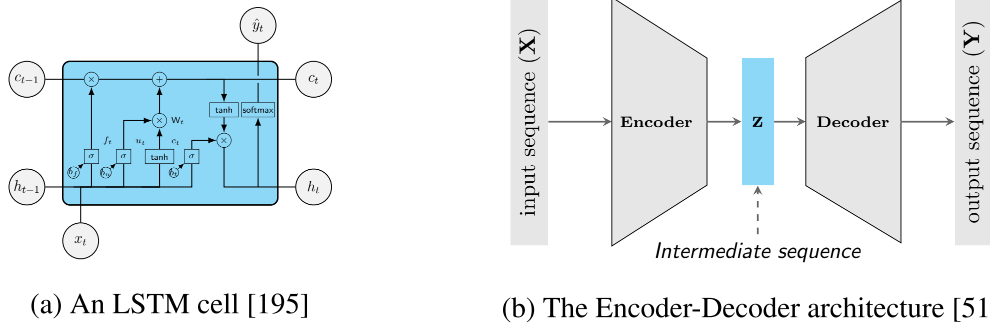
Threats to *internal validity* concern factors internal to our study that could have influenced the results. One possible threat can be seen through the results obtained for the datasets $\texttt{SH}_L$ and $\texttt{SH}_S$. As noted, these datasets exhibit lower precision/recall with respect to $\texttt{MV}_L$ and $\texttt{MV}_S$ due to the limited size of the training sets. However, these datasets were needed to compare FOCUS and PAM due to the limited scalability of PAM.

The main threat to *external validity* is that FOCUS is currently limited to Java programs. As stated in Section 4.1.2, however, FOCUS makes few assumptions on the underlying language and only requires information about method declarations and invocations to build the 3D rating matrix. This information could be extracted from programs written in any object-oriented programming language, and we wish to generalize FOCUS to other languages in the future.

4.2 LUPE

4.2.1 Machine translation with Encoder-Decoder

In the following, we describe the terminology used in the rest of the chapter, provide a motivating example for LUPE, and then recall the Encoder-Decoder architecture used in the proposed approach.



(a) An LSTM cell [195]

(b) The Encoder-Decoder architecture [51]

Fig. 4.11 LSTM and Encoder-Decoder.

Given an initial sequence of API invocations, we reformulate the problem of recommending APIs as predicting the sequence of additional invocations to be included. To perform such a task, we employ an Encoder-Decoder neural network, which has been conceptualized to solve the problem of machine translation [150, 253]. An Encoder-Decoder neural network is built on top of an atomic element called Long Short-Term Memory [253] (LSTM) unit. We briefly recall the LSTM technique before explaining how sequence-to-sequence learning can be used to deal with the problem of API recommendations.

▷ **LSTMs**. This supervised learning technique was proposed to learn better long-term dependencies by memorizing the input sequence of data [110]. Figure 4.11a depicts an LSTM cell, and described as follows. Given that i_t=$[h_{t-1},x_t]$ is the concatenation of h_{t-1} the hidden state vector from the previous time step, and x_t is the current input vector, then cell state c_t and hidden state h_t are propagated to the next cell. The output of the previous unit, together with the current input, is fed as the input data for a cell. The *sigmoid* and *tanh* functions defined as $\sigma(x)$=$(1+exp(-x))^{-1}$ and $\tanh(x)$=$2\cdot\sigma(2x)-1$, which are used to discard useless information and retain useful information. W_t, b_f, b_t, and b_u are the weight and bias matrices for different network entry, and hidden state matrix.

▷ **Sequence-to-Sequence learning**. An Encoder-Decoder neural network translates an input sequence X=$(x_1,x_2,\ldots,x_I)$ into an output sequence Y=$(y_1,y_2,\ldots,y_J)$, done by computing the conditional probability $P_\theta(Y|X)$ given below:

$$P_\theta(Y|X) = \prod_{j=1}^{J+1} P_\theta(y_j|Y_{<j},X) \tag{4.5}$$

where $P_\theta(y_j|Y_{<j},X)$ is the probability of creating the jth element of y_j, given $Y_{<j}$ and the input sequence X, where $Y_{<j}$ represents the output sequence consisting of characters from 1 to $j-1$. A *seq2seq* process is made of two phases: *(i)* producing a fixed size vector z from X, i.e., $z = f(X)$; and *(ii)* creating Y from z. In particular, z is generated from X using the following function: $z = \Delta(X)$, and then, Y is generated from z with the following function:

$$P_\theta(y_j|Y_{<j},X) = \Psi(h_j^{(t)}, y_j) \tag{4.6}$$

$$h_j^{(t)} = \Omega(h_{j-1}^{(t)}, y_{j-1}) \tag{4.7}$$

Ω is used to generate the hidden vector h_j, while Ψ calculates the probability of the one-hot vector y_j. The two functions are both recursive, where h_0 corresponds to z, and y_0 signals the beginning of one-hot vectors.

As shown in Figure 4.11b, an Encoder-Decoder LSTM is made of the following modules:

- *Encoder* is a group of LSTM cells and it gets as input a sequence X to produce the intermediate sequence z.
- *Intermediate sequence* is obtained through the encoding of information contained in X by *Encoder*.
- *Decoder* is a group of LSTM cells to generate the output sequence, accepting z as the input.

In the succeeding section, we will reformulate the API recommendation problem as a sequence-to-sequence learning one.

4.2.2 LUPE architecture

The LUPE architecture is depicted in Figure 4.12. Data is fetched from various open-source sources ① using the DATA CURATOR ② component. The collected data is then transformed into a suitable format to be stored in CSV files by the DATA CONVERTER ③, which then builds a universal dictionary containing APIs and their corresponding IDs. The SEQUENCE BUILDER ④ component uses the dictionary to extract APIs from Rascal M^3 [25] files. It also generates input for RECOMMENDER, which learns from data in the training phase to provide recommendations in the testing/deployment phase. The following subsections explain these components in detail.

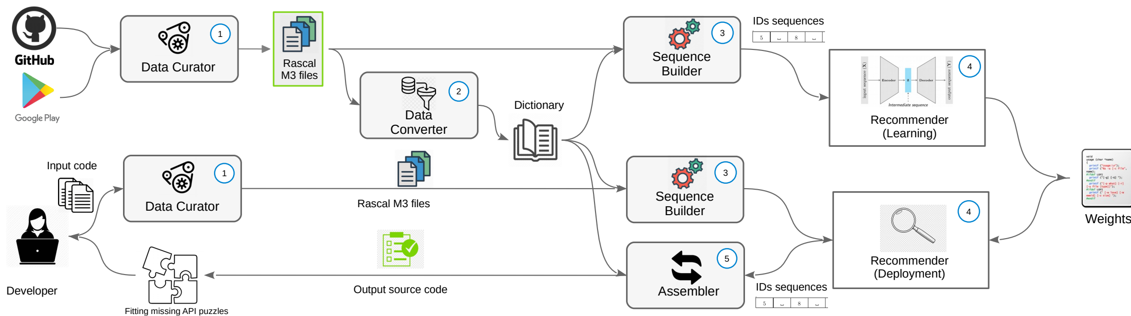


Fig. 4.12 System architecture.

4.2.2.1 Data Curator

This component crawls data from open-source repositories to build a training corpus for LUPE. It leverages the Rascal M^3 model [25] on source code to retrieve method definitions and API invocations.

M^3 is a relational and tree-based extensible and composable model for source code artifacts, with immutable value semantics, source location literals, and extensibility with additional types of binary relations. The model consists of two layers: an abstract syntax tree (AST) and a relationship layer. The former has a standard interface yet it is expected to be language dependent. The latter is more abstract, and suitable to be reused. These relationships represent fundamental aspects of a programming language's static semantics. In particular, in the Java extension binary relations take into account extra relations such as *extends*, *implements*, *methodInvocation*, *methodOverrides*, and *fieldAccess*, that are typically found between Java AST nodes. LUPE makes use of the *methodInvocation* relation to extract the relation between method definitions and API invocations. In this respect, the DATA CURATOR module builds program information including definitions, API invocations, field accesses (see Section 4.2.1). The relations of the M^3 model concerning $\texttt{declaration}_R$ and $\texttt{methodInvocation}_R$ contain an ordered list of pairs $\langle v_1, v_2 \rangle$, with v_1 and v_2 representing the declaration and the method invocation locations,[21] respectively. Since Rascal includes a grammar notation with built-in support for precedence and associativity of parse trees, the relation list of $\texttt{declaration}_R$ and $\texttt{methodInvocation}_R$ reflects the sequence of method invocations that occur in a method definition. These locations are uniform resource identifiers (URI) used to dictate logical locations of definitions and invocations. The URI locations resemble Java canonical names, where method definitions and invocations are represented as a compact string. This allows DATA CURATOR to identify the pointed resources without contextual information. A $\texttt{declaration}_R$ relation associates the logical location of a method with its URI location. Similarly, a $\texttt{methodInvocation}_R$ relation maps the artifact identities

[21] https://tutor.rascal-mpl.org/Rascal/Expressions/Values/Location/Location.html

of a *caller* to those of its *callee(s)*. Figure 4.13 depicts an instance of the *declaration/API invocation* pair extracted from Line 25 of the example in Figure 4.1b.

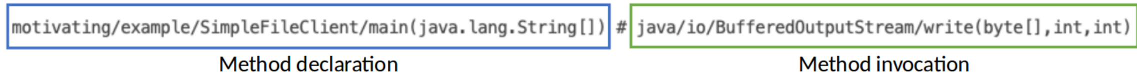


Fig. 4.13 Method declaration–Method invocation pair.

By analyzing open-source repositories, the DATA CURATOR component produces a set of Rascal M^3 files to feed as input for SEQUENCE BUILDER. In the training phase, it also submits the resulting M^3 files to DATA CONVERTER, which populates a universal dictionary as explained in the following subsection.

4.2.2.2 Data Converter

For each method definition, a point is set somewhere in the middle to break the definition into two parts. The first part is used as *input API invocations*, while the second one is the *output API invocations*, and they represent the upper and lower frames in Figure 4.1b, respectively. Such a point reflects different levels of maturity of a method. To get a well-trained model, it is necessary to consider all the possible breaking points within a declaration, i.e., we need to move the breaking point, invocation by invocation, and train the model accordingly.

To feed the sequence-to-sequence network, it is necessary to convert each definition into a chain of API invocations. First, every API is encoded with a unique number as follows. Then, given a training set, DATA CONVERTER counts the number of occurrences for each API to form a list of APIs and their frequency. The list is then sorted in descending order of frequency. Afterward, it traverses down the list from the top to assign an ID to each API. The IDs start from 0 and increase by one at every API. In this way, even when different APIs have the same frequency, they are assigned different IDs. Moreover, a more frequent API is assigned to a smaller number; altogether, this aims to avoid having long sequences, thus optimizing the computation.

For example, when analyzing a dataset collected from GitHub and Google Play containing a total of 102,459 API invocations (see Section 4.2.3), `java/lang/StringBuilder/append(java.lang.String)` is the most popular API, and thus is it assigned the ID=0. Meanwhile, `net/obry/ti5x/Main/findViewById(int)` is the least frequent API as it appears only once, thus it is assigned the ID=102458.

Table 4.7 Input, outputs APIs and their corresponding IDs.

	API	ID
Input APIs	java/io/FileOutputStream/FileOutputStream(java.lang.String)	5
	java/net/Socket/getInputStream()	8
	java/io/BufferedOutputStream/BufferedOutputStream(java.io.OutputStream)	12
	java/io/InputStream/read(byte[],int,int)	4
Output APIs	java/io/InputStream/read(byte[],int,int)	4
	java/net/Socket/Socket(java.lang.String,int)	15
	java/io/BufferedOutputStream/write(byte[],int,int)	23
	java/io/BufferedOutputStream/flush()	27
	java/net/Socket/close()	55
	java/io/FilterOutputStream/close()	81
	java/io/FileOutputStream/close()	41

After this step, a dictionary $\mathscr{D}$ to map between IDs and the real APIs is built to be used for further processing phases. As an example, Table 4.7 depicts the IDs[22] of the input and output API invocations representing the upper and lower frames in Figure 4.1b obtained by looking up $\mathscr{D}$.

4.2.2.3 Sequence Builder

Starting from Rascal M^3 files as input, SEQUENCE BUILDER ④ renders each input definition d_i into s_i – the sequence consisting of only IDs, by looking up the dictionary, i.e., $d_i \xrightarrow{\mathscr{D}} s_i$, $i = \overline{1..N}$, where N is the total number of definitions. Figure 4.14 shows how the input and output sequences in Table 4.7 are built. The resulting ID sequences are as follows: X="*5␣8␣12␣4*" and Y="*4␣15␣23␣27␣55␣81␣41*", where the "␣" character represents a space and it is used to distinguish between the IDs within a definition.

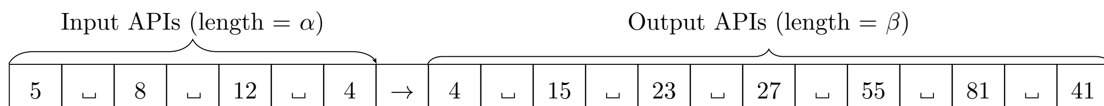


Fig. 4.14 Input and output API sequences for the example in Figure 4.1b and Table 4.7.

Since the σ and tanh functions (see Section 4.2.1) accept only binary numbers as input, the next step is to convert all the IDs to vectors containing 0 and 1. First, C_v — the corpus of all the possible characters used to form the IDs — is built. It contains 11 letters, i.e., $C_v = \{0, 1, \ldots, 9, \}$. Then, each character is encoded using a one-hot vector whose length corresponds to Π, the corpus's number of characters, i.e., Π=$|C_v|$=11. Moreover, we also compute the length of all sequences to find Θ – the maximum length. Afterward, a sequence

[22] The IDs are for illustration purposes only. In practice, they might be longer. For the sake of presentation, we avoid using such numbers throughout the chapter.

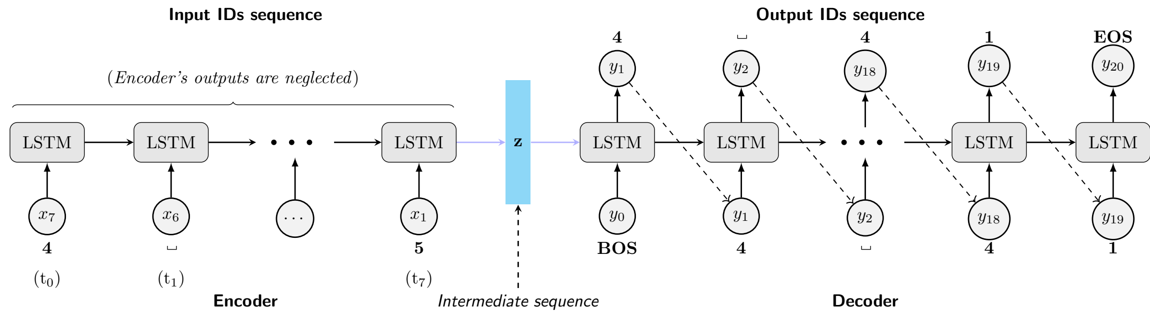
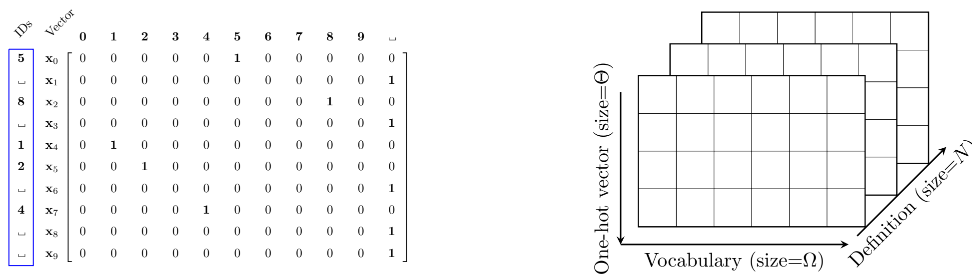


Fig. 4.15 Training with reversed input sequence X="*4␣21␣8␣5*" and output sequence Y="*4␣15␣23␣27␣55␣81␣41*".



(a) One-hot matrix parsed for the input sequence in Table 4.7

(b) Tensor to store input data for a complete training set

Fig. 4.16 Data encoding with matrix and tensor.

is represented as a 2D matrix of size $\Theta \times \Pi$, where each row corresponds to the one-hot vector of a character within a sequence. A sequence having a length smaller than Θ, is padded with "␣" to fill the gap.

Figure 4.16a shows how the X="*5␣8␣12␣4*" input sequence in Table 4.7 is transformed into the corresponding one-hot matrix. Each row represents a one-hot vector of a specific character in the sequence. Let Θ be 10,[23] since $len(X) = 8 < \Theta$, then the last two rows x_8 and x_9 are filled with "␣".

By transforming all definitions of a collected dataset, we obtain a tensor of size $\Theta \times \Pi \times N$ as shown in Figure 4.16b. Each slide corresponds to a definition, the columns are the vocabularies used to form the IDs, and the rows represent one-hot vectors. The tensor is used as input for the recommendation engine, which is described in the following subsection.

[23] Again, such a small number is used to ease the presentation. In general, the maximum sequence length Θ is large.

4.2.2.4 Recommender

In the training phase, the RECOMMENDER component learns from data by translating an input sequence into an output sequence, resembling machine translation activities [150, 253]. As in the motivating example of Section 4.2.1, the aim is to learn to generate the output IDs sequence *Y*="*4␣15␣23␣27␣55␣81␣41*" from the input IDs sequence *X*="*5␣8␣12␣4*", and the process is illustrated in Figure 4.15. Following the Encoder-Decoder paradigm described in Section 4.2.1, ENCODER consists of a stack of LSTM units to encode input sequences. According to an empirical work [253], training in the reverse order of the input sentence fosters a better prediction performance. Thus, every input sequence is inverted before being fed. At each time step, only one one-hot vector representing a character is introduced to ENCODER. For instance, at time step t_1, the one-hot vector of the character "␣" is provided as input.

Similarly, DECODER is made of LSTMs to decode the input sequence. At each time step, the label vectors representing output sequences are also introduced singularly. Moreover, the output of a time step is fed as input to the next one. `[BOS]` and `[EOS]` are two special characters being padded to signal the beginning and end of every output sequence, respectively. Once the training phase is completed, the Encoder-Decoder network returns a set of weights, which can be used independently from the training data.

In the testing phase (i.e., at deployment time), RECOMMENDER uses the obtained weights to generate the output sequence, given an input one. Such a phase is much faster compared to training, as there is no need to iteratively traverse the data, i.e., the inference is done only once for every input.

4.2.2.5 Assembler

The predicted sequences consisting of characters representing IDs returned by RECOMMENDER ④ in the testing phase are fed as input to ASSEMBLER. The latter performs the inverse mapping to convert the IDs back to the form of APIs using the dictionary, i.e., $s_i \xrightarrow{\mathscr{D}} d_i$, $i = \overline{1..N}$. This mapping results in a sequence of APIs, which is supposed to fit the missing puzzle as shown in Figures 4.1a and 4.1b. The ASSEMBLER component adopts an existing technique [177] to retrieve source code as follows. It combines the input code with the recommended APIs to form a query and searches the whole training corpus for relevant definitions, which contain as many common APIs of the query as possible. If there are many retrieved definitions with the same number of common APIs, then it is necessary to choose the right one. As LUPE considers the order in which the API calls occur, and thus if we find definitions with the same number of APIs as well as the order in which they appear,

then they should implement the same functionality, or in other words, it is assumed that they are the same definition. Essentially, we can pick one of them as the final recommendation. Afterward, the real source code of a method definition is extracted by leveraging the computed M^3 model and the project location. Finally, the resulting code snippet is returned as the recommendation.

4.2.3 Evaluation

4.2.3.1 Research questions

The *goal* of this study is to evaluate the performance of LUPE on real-world datasets and to compare it with GAPI, a state-of-the-art API recommender. The evaluation is performed in the *context* of Android development. In particular, we address the following research questions:

- **RQ**$_1$: *To what extent is LUPE able to recommend relevant API invocations with different training configurations?* Using an Android dataset, we study how well LUPE recommends relevant APIs to developers under different configurations, involving different training sets of different sizes.
- **RQ**$_2$: *How does LUPE compare with* GAPI*?* We evaluate LUPE by considering two state-of-the-art approaches in API recommendation, i.e., GAPI [145], and FACER [9]. Both the evaluations have been conducted on the **DS**$_2$ dataset described in Section 4.2.3.
- **RQ**$_3$: *How does LUPE compare with* FACER*?* Using a curated dataset of JAR artifacts, we study how well LUPE recommends relevant APIs to developers under different configurations, i.e., if it is able to gain a better performance when being fed with well-curated input data.

4.2.3.2 Baselines and datasets

The rationale behind the selection of GAPI [145] and FACER [9] as baselines are as follows. The former has been found to outperform other API recommenders such as FOCUS [177] – a well-established baseline – with respect to various quality metrics. Meanwhile, the latter is a recent approach that employs the Lucene indexing engine in combination with a frequent pattern mining strategy to retrieve relevant API calls. More importantly, FACER has been thoroughly evaluated by means of well-defined user studies. Altogether, this makes GAPI and FACER suitable baselines that LUPE should be compared with.

GAPI exploits the high-order relationships between projects and API function calls by using three different modules, i.e., graph construction, graph embedding, and API usage prediction. The first module is used to parse the project source code and extract the relevant information. Then, the Graph embedding module encodes such data in a graph-based structure by exploiting two different embedding techniques, i.e., Gated Recurrent Units (GRU) and Stacked Graph Convolution. GAPI eventually retrieves the most similar API usages compared to the input ones. Therefore, we consider GAPI as the state-of-the-art approach which LUPE should be compared to.

We evaluate LUPE against GAPI using a real-world dataset to showcase the advantages of our approach. Specifically, we focus on Android-related recommendations, which may be useful to support mobile developers. To mine data, we made use of an existing technique [177] as follows. First, we looked for open-source projects with the *Android-TimeMachine* platform [91], fetching apps and their source code from Google Play[24] and GitHub. Then, APK files are retrieved from the Apkpure platform,[25] using a Python script. Afterward, APK files are converted into the JAR format utilizing *dex2jar* [1]. The JAR files were then fed as input for Rascal, which renders them into the M^3 format [25]. We curated the first dataset of 1,200 apps (DS_1), and for the comparison with GAPI, we crawled the second dataset (DS_2), which is smaller than DS_1 and contains 200 apps of average size. A summary of the datasets, reporting the number of method declarations, invocations, as well as the total size of the datasets in the M^3 format, is shown in Table 4.8.

Table 4.8 Datasets.

Name	# of artifacts	# definitions	# invocations	Total size
DS_1	1,200	447,686	102,459	807 MB
DS_2	200	4,346	9,069	132 MB
DS_3	4,823	316,138	4,956,290	837 MB
DS_{3S}	3465	170,277	2,618,273	837 MB

Compared to LUPE and GAPI, FACER employs a different recommendation engine. To retrieve relevant API calls, it uses Lucene together with a frequent pattern mining strategy. After the indexing phase, it employs the so-called Method Clone Structure (MCS) detection to find similar invocations in method clone groups. Apart from API calls, FACER can recommend relevant features for the active context given as the query, i.e., the method's body. Since LUPE retrieves API function calls without suggesting the entire snippet, we do not consider this second type of recommendation to make the comparison as fair as possible.

[24] https://play.google.com/
[25] https://apkpure.com/

Similarly, we ran FACER on the same dataset, i.e., $\mathbf{DS}_2$, by obtaining the corresponding GitHub repositories for each .APK file. Afterward, we ran the FACER analyzer component on them and we obtained 180 projects due to some exceptions in the parsing phase. From this list, the tool extracts 53,836 methods and 117,046 API function calls. It is worth noting that FACER retrieves results for 5,549 methods using the MCS detection strategy. Amongst these, 3,328 produces a list of API function calls that we used in the comparison with LUPE. Meanwhile, for the left part, i.e., 2,221 queries, FACER recommends just the method features. Thus, we discarded these contexts from the evaluation since LUPE does not support this kind of recommendation at its current state of development.

To evaluate the performance of LUPE with a different dataset consisting of Maven artifacts,[26] we curated a dataset from the Maven Dependency Graph (MDG) [29]. MDG is an open-source dataset that uses a graph database to record Maven artifacts together with all of their relationships, including upgrades and dependencies. $\mathbf{DS}_3$ consists of 4,823 artifacts mined from MDG by selecting the ones that use at least two dependency relationships to the ten most popular libraries.[27] This constraint allows us to have a dataset where the artifacts share common API function calls.

In order to understand the impact of infrequent method definitions on LUPE, $\mathbf{DS}_{3S}$ is a dataset extracted from $\mathbf{D}_3$ consisting of 3,990 artifacts that contain at least a method definition that exactly occurs in at least 10 extracted artifacts.

4.2.3.3 Settings

The hardware and software configurations of the platforms used for experimentation are listed in Table 4.9. We trained LUPE using $\mathbf{P}_1$, a powerful platform dedicated to run DL models. The weights obtained by the training were saved in external files, which are used independently from the training data. To perform testing, we uploaded the weights to both Google Colab[28] and $\mathbf{P}_2$ and ran the inference separately. This simulates a real deployment scenario, where training and testing might be conducted in different environments. While training can be done on powerful servers, testing should run on developers' computers with limited hardware resources, because it is where the trained tool should run in a real usage scenario, i.e., when a developer uses it.

To evaluate LUPE, we adopted the ten-fold cross-validation technique [287]. Thus, within the training data, 80% and 20% are used for training and validation, respectively. The validation set was used to tune the LUPE hyperparameters, among others, the number of

[26] https://mvnrepository.com
[27] https://mvnrepository.com/popular?p=1
[28] https://colab.research.google.com/

Table 4.9 Experimentation settings.

Name	Training (P_1)	Testing (P_2)
RAM	96 GB	32 GB
CPU	Intel® Xeon CPU E5-2678 V3 @ 2.50GHz × 24	AMD@ Ryzen 7
GPU	NVIDIA GeForce GTX 1080Ti	NVIDIA GeForce GTX 2060
OS	Ubuntu 20.04	Windows 10
Python	3.7.5	3.7
TensorFlow	2.6.0	2.6.0
Numpy	1.15.4	1.16.3
Timm	0.3.1	–
Gensim	3.7.1	3.7.1

hidden units used in each LSTM cell.[29] By an empirical evaluation, we set the number of epochs to 100 and batch size to 500. In contrast, the hold-out cross-validation technique was used to compare LUPE with GAPI, i.e., 80% and 20% of the dataset are for training and testing, respectively, aiming for a fair comparison with the baseline. We used the hyperparameters that achieve the performance according to the GAPI original evaluation [145]. In particular, the dimension of the linear layer was set to 128, and the graph convolution kernel size to 64. The maximum length of the text attribute and the number of graph convolution layers were set to 10.2 and 0.5, respectively. Concerning the optimizer, GAPI makes use of the Adam model to train with the learning rate 5×10^{-4} and weight decay. The batch size and negative sample size were set to 2 and 64, respectively. We ran GAPI with 40 epochs, as the model reaches a stable accuracy after being trained with this number of iterations.

Thanks to its internal design, LUPE is a definition-based, rather than a project-based approach, i.e., the input data to feed as input to the recommendation engine is a single definition, not a set of definitions. This is different from most existing approaches [83, 145, 177], where apart from the testing definition, additional definitions must be incorporated as context information. Thus, to evaluate LUPE for each project, we randomly chose a certain number of definitions for testing. Every time, only a testing definition is fed as input to LUPE, and for this testing definition, the first half of invocations is used as a query. The remaining invocations are removed and saved as ground-truth data, simulating the scenario in Figures 4.1a and 4.1b, where the developer has finished the first part of the code and expects to get suggestions for the second part. As shown in Section 4.2.2, the point to break a definition into two parts can be flexibly configured, reflecting different levels of completeness of the definition.

[29]It is worth noting that the number of hidden units is not equal to the number of LSTM cells shown in Figure 4.15. There are only two LSTM cells in Figure 4.15, one for Encoder and one for Decoder, but they are shown in different time steps.

If the input definition has a small number of invocations, then LUPE is not able to produce any predictions. As a result, in the evaluation we had to set a threshold for the number of APIs larger than 10. This led to a decrease in the number of definitions suitable for the evaluation. Before running the experiments, we read every project in the considered datasets, and saw that most of the projects have a considerably low number of method definitions. Therefore, upon addressing RQ1, for each project, we randomly select k=10 method definitions for testing. We consider the following configurations. For each testing definition $\mathbf{d}_t$, given that $\mathscr{S}_t$=size($\mathbf{d}_t$) is the total number of API invocations of $\mathbf{d}_t$, α=round($\mathscr{S}_t/3$) and α=round($\mathscr{S}_t/2$) invocations are used as query, corresponding to Configurations $\mathbf{C}_1$ and $\mathbf{C}_2$, respectively. The remaining invocations are saved as ground-truth data. Finally, we consider a further configuration, $\mathbf{C}_3$ in which we select, in the testing data, only definitions that appear at least a certain number of times f in the training data. In practice, this means that LUPE has properly learned these definitions during the training phase. For $\mathbf{C}_3$, we set f=10.

Being a data-driven approach, LUPE heavily relies on the input API calls to function. Thus, if there is not enough data to feed the prediction engine, LUPE cannot produce any recommendation. This is the case of method declarations with a few lines of code, or those without a body. In the evaluation, we avoided considering these definitions by setting a threshold for the number of API calls. In particular, method definitions are used for testing only when they contain more than a certain number of API calls, i.e., 10 method invocations.

4.2.4 Results

In this section, we report results by addressing the two research questions formulated in Section 4.2.3.

$\mathbf{RQ}_1$: *To what extent is LUPE able to recommend relevant API invocations with different training configurations?* Table 4.10 reports the success rates obtained for all the ten folds for the three configurations $\mathbf{C}_1$, $\mathbf{C}_2$, and $\mathbf{C}_3$. Overall, it is possible to notice that the scores are always larger than 0.90 by all the folds, with 1.00 being the maximum value. This suggests that LUPE retrieves at least a matched invocation for almost all the testing definitions.

Table 4.10 LUPE Success rate.

Conf.	**F01**	**F02**	**F03**	**F04**	**F05**	**F06**	**F07**	**F08**	**F09**	**F10**	**Avg.**
$\mathbf{C}_1$	0.980	0.980	0.970	0.979	0.972	0.972	0.960	0.975	0.972	0.970	0.973
$\mathbf{C}_2$	0.900	0.942	0.944	0.940	0.900	1.000	0.955	0.964	0.929	0.951	0.942
$\mathbf{C}_3$	0.981	1.000	0.999	0.996	0.995	1.000	1.000	1.000	0.996	1.000	**0.997**

For $\mathbf{C}_1$, we show the precision and recall scores using violin plots, i.e., combining boxplots and density traces [109], in Figure 4.17 and Figure 4.18, respectively. As it can be seen, a large part of the scores piles up on the upper part of the diagram, i.e., in the $[0.9,\ldots,1.0]$ range. This means that most of the testing definitions get well-matched API invocations.

However, many points are in the whiskers, towards the 0.0 level, which corresponds to a poor performance. To investigate the distribution of the scores, for each metric, either precision or recall – being called with a common name m – we combine the scores from all the folds and display them together in the stacked bar charts in Figure 4.19. There are the following three levels: *Low*: $m \leq 0.5$, *Medium*: $0.5 < m \leq 0.9$, and *High*: $0.9 < m \leq 1$. The metrics in Figure 4.19 confirm the outcome in Figures 4.17 and 4.18, i.e., by more than 90% of the definitions, LUPE gets high precision and recall, being greater than 0.9.

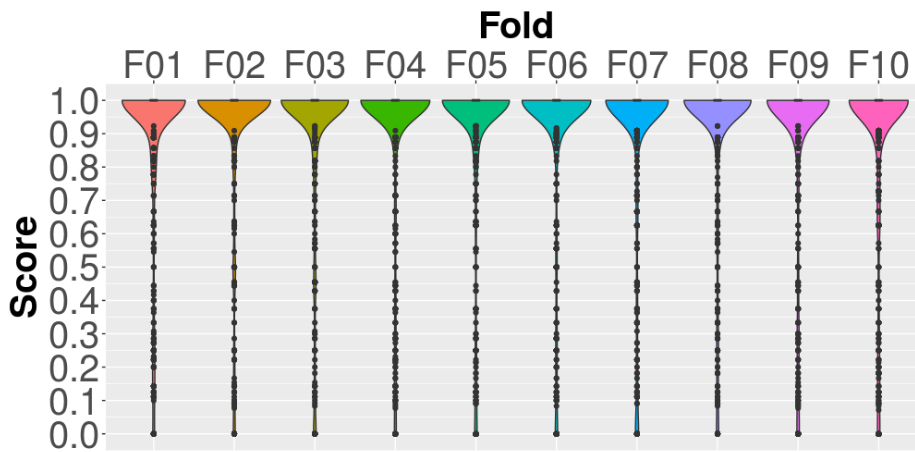


Fig. 4.17 LUPE Precision with configuration C_1.

We then investigate how the length of the input data impacts the prediction performance by considering $\mathbf{C}_2$, where half of each testing declaration is used as a query, i.e., α=round($\mathscr{S}_t/2$), and the rest is ground-truth data. The final results are shown in Figure 4.20. As the figure shows, LUPE yields a high prediction accuracy for most of the testing method definitions. Compared to the results for $\mathbf{C}_1$ in Figure 4.19, by $\mathbf{C}_2$, LUPE improves its performance, especially with the precision scores. *We conclude that being fed with an increasing number of API invocations, LUPE returns better predictions.*

Finally, we investigate whether a further augmentation of the training data (i.e., Configuration $\mathbf{C}_3$) is helpful. Figures 4.21 and 4.22 show the precision and scores obtained for all the ten folds.

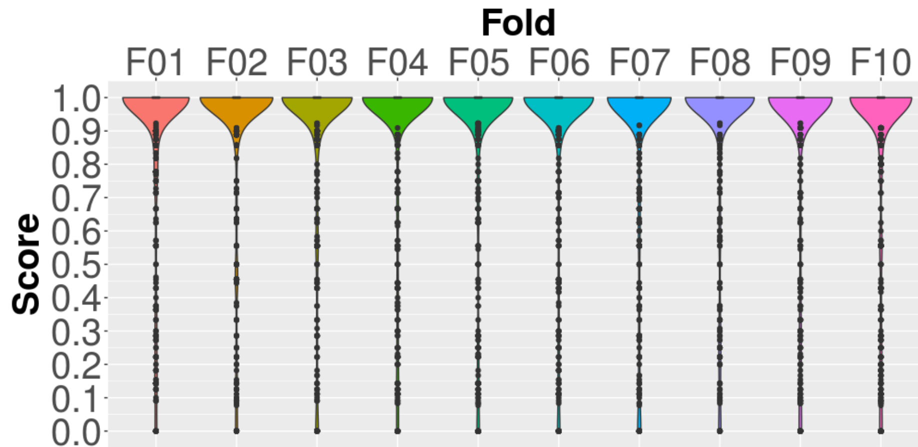


Fig. 4.18 LUPE Recall with configuration C_1.

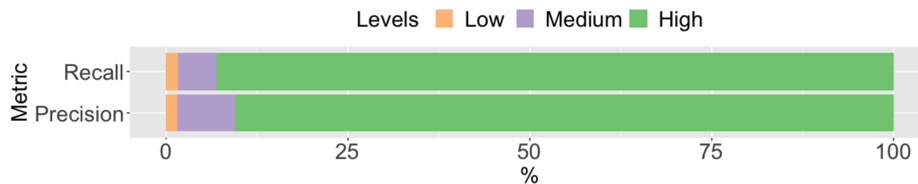


Fig. 4.19 Combination of the scores from all the folds with configuration C_1.

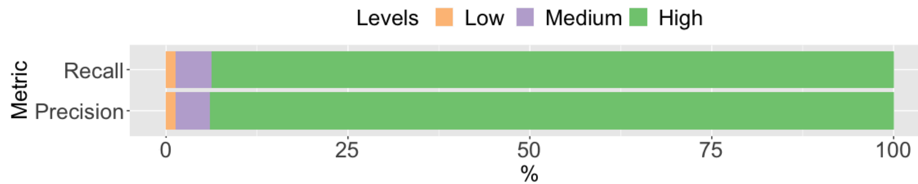


Fig. 4.20 Combination of the scores from all the folds with configuration C_2.

As it can be seen, compared to the scores in Figures 4.17 and 4.18, the results obtained with $\mathbf{C}_3$ are substantially improved. In particular, most of the precision and recall scores reside on the upper part of the diagrams – starting from the 0.95 level – corresponding to precise predictions. Moreover, the number of points in the lower parts of the whiskers decreases, compared to those in Figures 4.17 and 4.18.

This implies that if LUPE is trained with more frequent method definitions, it can return more correct predictions. In this respect, the augmentation done by cloning method definitions and planting them in different parts of the training data might help LUPE to achieve a better prediction performance.

The precision and recall metrics measure how well a set of recommended API invocations matches the ground truth data. However, they do not reflect if the two sets also match the

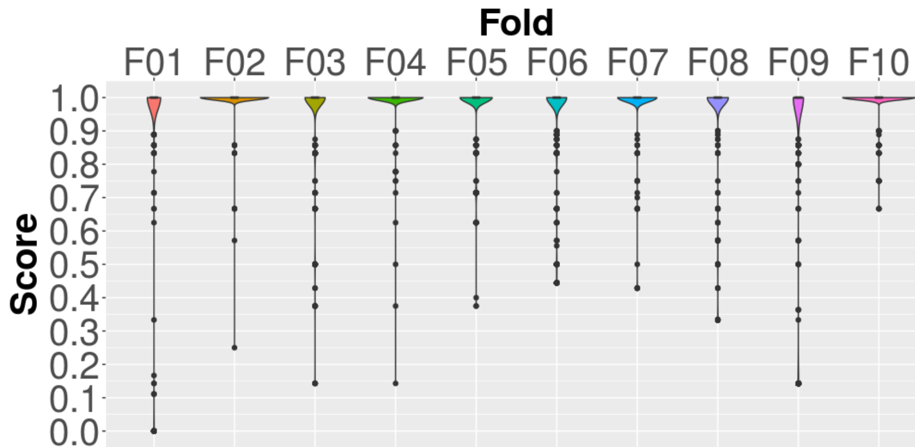


Fig. 4.21 LUPE Precision with configuration C_3.

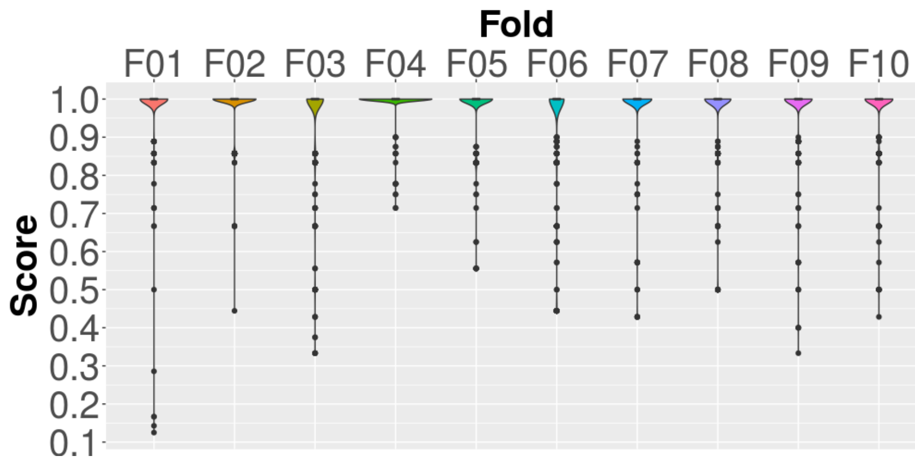


Fig. 4.22 LUPE Recall with configuration C_3.

Table 4.11 Percentage of definitions getting 0 as Levenshtein distance (%).

F01	**F02**	**F03**	**F04**	**F05**	**F06**	**F07**	**F08**	**F09**	**F10**
75.67	82.00	78.41	75.83	78.33	79.66	81.44	80.00	75.66	75.00

order in which the API invocations occur. To this end, we measure the similarity between two sequences of API invocations using the Levenshtein distance [140]. Such a metric measures the similarity between two invocations chains, taking into account the pairwise order of invocations; a distance of 0 corresponds to a perfect match. For the sake of brevity, we consider only Configuration $\mathbf{C}_3$ and report in Table 4.11 the percentage of method definitions getting a Levenshtein distance of 0. By all the folds, at least 75% of the definitions get recommendations that perfectly match the ground-truth data.

Answer to RQ_1. LUPE achieves high (overall > 0.9) precision and recall, returning at least an exact match for each test data point. Its performances further improve once it gets more training data for a given method definition. Finally, LUPE can recommend a perfect match, also concerning the order of invocations.

RQ_2: *How does LUPE compare with* GAPI*?* In this research question, we compare LUPE with the two mentioned approaches, i.e., GAPI and FACER. Concerning the comparison with GAPI, we analyze the performance of the two tools with respect to *(i) timing efficiency*, i.e., whether they can support a real-world setting, where there is supposedly limited computational resource; and *(ii) effectiveness*, i.e., if they can provide accurate recommendations. For this comparison, we experimented with Configuration C_2 for both GAPI and LUPE, as it reflects a more mature method definition, where the developer has already finished half of the code and needs recommendations related to the remaining half of the definition. With respect to the comparison between LUPE with FACER, due to FACER's internal design, it is not possible to replicate the traditional ten-fold validation since the API function calls are stored in a relational database with several integrity references among projects, methods, and API calls. Aiming for a fair comparison, we opted for a validation set similar to the experiments conducted in the original paper [9]. As described in Section 4.2.3, we fed FACER with the projects in $\mathbf{DS}_2$, obtaining 3,328 suitable testing methods that have been considered also by LUPE. We eventually compared the two approaches in terms of *effectiveness*, avoiding the *timing efficiency* comparison. The rationale behind this choice is due to the fact that FACER and LUPE rely on completely different working mechanisms, thus requiring diverse running platforms that are not comparable.

4.2.4.1 Comparison with GAPI

▷ **Timing efficiency.** We realized that GAPI suffers from prolonged training, i.e., it takes a long time to learn, even on a small set of data. In particular, training GAPI with the $\mathbf{DS}_2$ dataset (200 projects of average size, see Section 4.2.3) on the $\mathbf{P}_1$ computer (see Table 4.9) requires 62 minutes to reach a stable performance, i.e., after 40 epochs. Training GAPI with $\mathbf{DS}_1$ is almost impossible, even on $\mathbf{P}_1$ – a powerful server – since the required time soars. In contrast, the execution of LUPE is much faster, even for a large dataset, i.e., training LUPE with $\mathbf{DS}_1$ and $\mathbf{DS}_2$ on $\mathbf{P}_1$ requires 30 and 4 minutes, respectively, to get a stable accuracy. Thus, to ease the comparison between LUPE and GAPI, we opt for $\mathbf{DS}_2$, instead of $\mathbf{DS}_1$. We measure the testing time of both systems, benchmarking with $\mathbf{P}_2$. Such a computer is utilized since it represents a more popular execution environment.

The measurements show that GAPI is very fast, i.e., it needs only 0.02 seconds to produce a recommendation. Meanwhile, on average LUPE spends 1.2 seconds for the same task. We conclude that, *compared to LUPE,* GAPI *is much more efficient in testing*. Nevertheless, *LUPE is more efficient in training than* GAPI, also because LUPE generates a complete chain of invocations at a time.

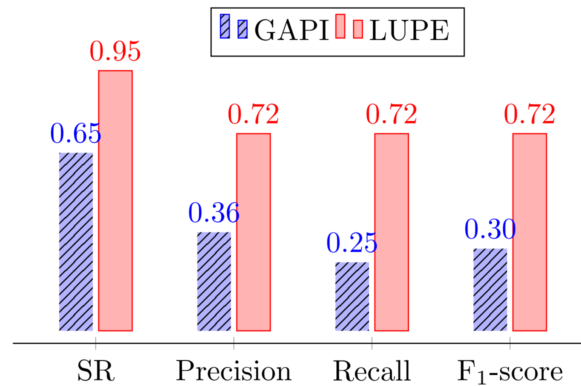


Fig. 4.23 Comparison between GAPI and LUPE.

▷ **Effectiveness.** As done in the original work [145], we use the hold-out validation method [287] for running both GAPI, and LUPE, i.e., 80% and 20% of the dataset are used for training and testing, respectively. Since GAPI recommends a ranked list of APIs, while LUPE provides a complete sequence of APIs, we have to fix the number of items for GAPI, i.e., N=10. This also means that we cannot represent the results using the widely adopted precision-recall curves [145, 177], where N needs to be varied. For each testing project, k=10 method definitions are selected for testing. We compare the two systems by averaging the best scores obtained in different trials. The final results are reported in Figure 4.23.

Overall, it is evident that, compared to GAPI, LUPE gets a better prediction performance for all the considered metrics. In particular, GAPI gets 0.65 as success rate, while the corresponding value obtained by LUPE is 0.95. More importantly, LUPE gets much better precision, recall, and F_1 scores. The average performance of GAPI is 0.36 for precision, 0.25 for recall, and 0.30 for F_1 score. LUPE achieves 0.72 for all three metrics.

4.2.4.2 Comparison with FACER

As discussed in Section 4.2.3, we cannot directly compare LUPE with FACER since the two approaches rely on completely different techniques, i.e., machine translation and

frequent pattern mining, respectively. Hence, this section reports the results obtained by FACER considering the same dataset used in the former comparison, i.e., DS_2. Table 4.12 summarizes the average values for each metric computed on the 3,328 methods that produce at least a set of recommendations.

Table 4.12 Results obtained by FACER on DS_2.

Metric	**Average value**
Success rate	0.11
Precision	0.04
Recall	0.05
F_1 score	0.03

It is evident that FACER obtains very low results by all the measured metrics, i.e., all the average values are below 0.10 apart from success rate. In particular, the success rate is just 0.11, meaning that FACER is not able to support the method extracted from $\mathbf{DS_2}$. This is further confirmed by the precision, recall, and F1 scores that are always smaller than 0.06. These results can be explained by *(i)* the composition of the dataset and *(ii)* the different evaluation methodology conducted in the original study. Concerning the employed dataset, FACER is capable of extracting a total number of 53,836 methods but only 3,328 of them produce a non-empty list of API function calls as discussed in Section 4.2.3 even though we properly followed all the required steps.[30]

Concerning the original validation, the approach was assessed by combining an automated evaluation and a user study [9]. The former has been used to set up the algorithm parameters that lead to better accuracy, i.e., intra-group method similarity and minimum support, by running 20 queries manually selected. By the latter, the authors involved a total number of 49 participants with different backgrounds. In the scope of our work, we evaluate FACER using our dataset and the identified metrics, i.e., success rate, precision, recall, and F1 score. Altogether, we see that LUPE outperforms FACER in terms of recommended API function calls on the given dataset, by referring to the results obtained in the previous analysis.

Answer to $\mathbf{RQ_2}$. While GAPI is more timing efficient in the testing phase, LUPE is much more timing efficient in training. More importantly, LUPE outperforms GAPI in success rate, precision, recall, and F_1 score. FACER achieves the worst performance, even though a comprehensive comparison cannot be conducted due to the different nature of the approaches.

[30]To avoid any bias in the replication process, we contacted the corresponding author of the original paper (to whom we would like to express our gratitude), who kindly provided us with the replication package, and assisted us to run the whole FACER pipeline, thus guaranteeing the correctness of the process.

RQ$_3$**:** ***How does LUPE compare with*** **FACER?** To evaluate whether LUPE yields a good performance considering the recommendations in different development domains, we run the tool on **DS**$_3$ and **DS**$_{3S}$. It is worth noting that a preprocessing step has filtered out infrequent API function calls from **DS**$_3$ to obtain **DS**$_{3S}$@. The success rate, the ρ_L, and training and testing time obtained for **DS**$_3$ and **DS**$_{3S}$@ are reported in Tables 4.13 and 4.14, respectively. It is evident that the best performance is achieved on a curated dataset where more similar method definitions are present, i.e., **DS**$_{3S}$@. For instance, the best success rate for **DS**$_3$ and **DS**$_{3S}$@ is 0.545 and 0.992, respectively. The analysis of ρ_L further supports this conclusion. For example, the best ρ_L for **DS**$_3$ is 2.6, while **DS**$_{3S}$@ reaches 88.5. The violin plots of Figures 4.24 and 4.25 provide a more informative representation of the shape of the distribution, allowing us to better understand the density's magnitude. The reported figures indicate that LUPE achieves high precision and recall scores only on the **DS**$_{3S}$@ dataset, while the results are not good enough when **DS**$_3$, a more heterogeneous dataset, is considered.

Table 4.13 **RQ**$_3$: Success rate and percentage of recommendations getting 0 as Levenshtein distance (ρ_L) on **DS**$_3$.

	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10
success rate	0.496	0.510	0.517	0.496	0.496	0.545	0.492	0.538	0.490	0.529
ρ_L %	0.5	0.4	0.4	0.4	0.7	0.3	0.7	0.4	1.5	2.6
Training time (s)	7,358	5,832	5,917	5,913	5,367	5,911	5,918	8,489	5,909	5,904
Testing time (s)*	21,489	24,531	26,747	26,107	15,367	35,973	17,668	12,089	32,843	25,343

[1] *Testing time considering more than 15K queries.

Table 4.14 **RQ**$_3$: Success rate and percentage of recommendations getting 0 as Levenshtein distance (ρ_L) on **DS**$_3$.

	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10
success rate	0.895	0.916	0.992	0.794	0.960	0.984	0.974	0.981	0.990	0.986
ρ_L %	66.5	40.9	88.5	43.5	76.6	91.2	83.4	73.9	87.6	79.5
Training time (s)	4,288	2,893	2,885	2,786	5,367	2,885	2,891	8,489	2,888	2,893
Testing time (s)*	11,711	9,652	9,342	9,542	5,367	11,639	7,323	8,489	9,678	12,772

[1] *Testing time considering more than 5.7K queries.

Tables 4.13 and 4.14 report, for both **DS**$_3$ and **DS**$_{3S}$@, the training and testing computed by **P**$_1$ and **P**$_2$ respectively. The testing time corresponds to computing 15K and 5.7K recommendations once the recommendation engine has been trained. For instance, LUPE needs a maximum of 32,843 and 12,772 seconds to perform inference for all 15K and 5.7K queries respectively. In particular, with **DS**$_3$ and **DS**$_{3S}$@, given recommendation request the average time of predicting the next sequence of API function calls is 32,843/15,000 = 2.2 seconds and 12,772/5,000= 2.55 seconds. The preprocessing steps are beneficial to the

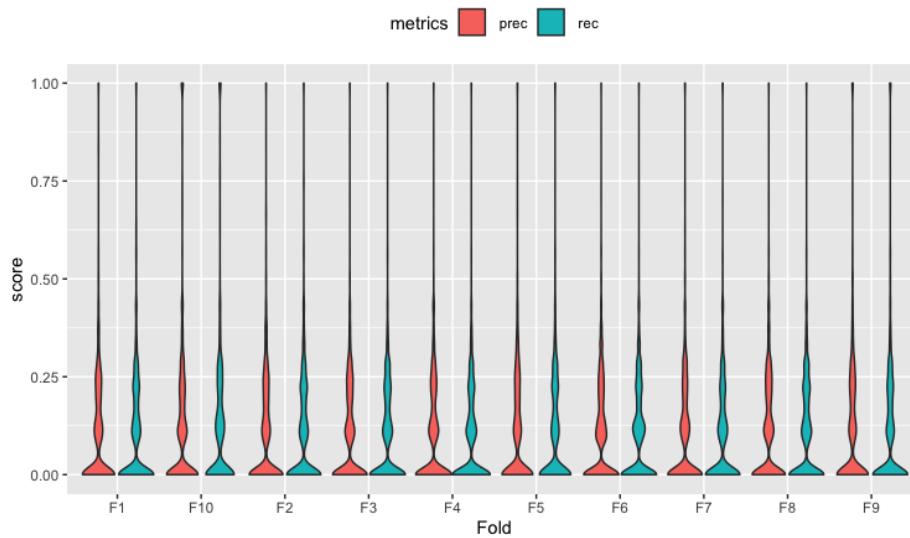


Fig. 4.24 LUPE Precision recall on $\mathbf{D}_3$.

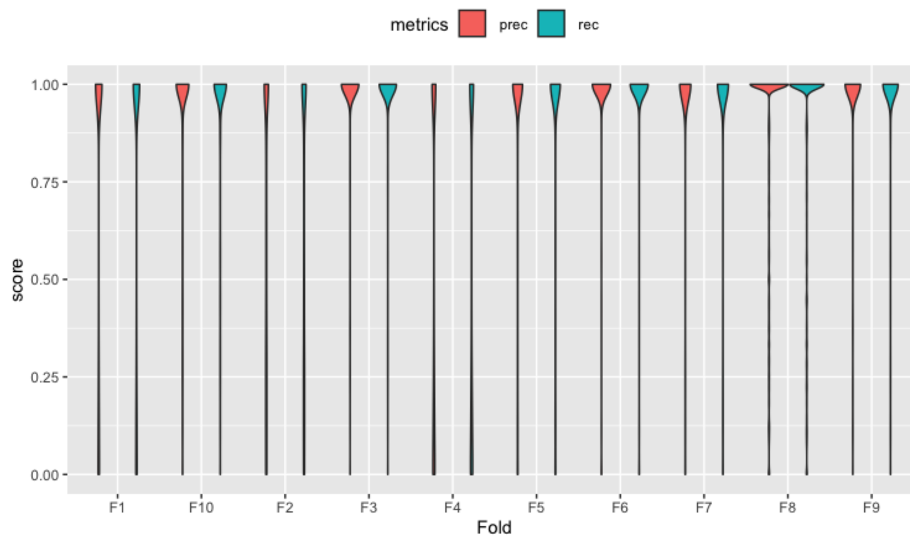


Fig. 4.25 LUPE Precision recall on $\mathbf{D}_{3S}$.

recommendation process as they allow LUPE to retrieve more relevant API function call sequences, thus improving the recommendation performance.

Answer to RQ$_3$. LUPE can considerably improve its performance on a well-curated dataset, i.e., where there are more popular method definitions. Though the training step is time consuming, it can be carried out on powerful computing platforms that export weights for usage by computers with lower processing demands. Instead, once the model has been trained, LUPE can provide recommendations in just few seconds.

4.2.4.3 Discussion

Differently from other recommenders, such as PAM [83] or FOCUS [177], LUPE does not require the whole training set to generate a recommendation, but only the model and API dictionary. This is a practical advantage of LUPE when deploying it on developers' machines, as the training set can be very large in size (e.g., $\mathbf{DS}_1$ with 807 MB as the total size, see Table 4.8).

To infer recommendations, LUPE only needs a testing definition as the input data, without considering other remaining definitions, as is the case with many other existing API recommenders, including UP-Miner [276], PAM [83], FOCUS [177], or even GAPI [145]. In other words, rather than being *project-based*, LUPE is a *definition-based* approach. This is indeed an advantage since, in practice, the tool can provide recommendations, even when there are no additional method definitions to be used as context.

To the best of our knowledge, LUPE is the only tool that uses the sequence-to-sequence deep learning mechanism to recommend API usage. This paves the way for further improvement by employing well-founded techniques conceived for the machine translation domain. We anticipate that the application of transfer learning [283] and attention techniques [271] might increase efficiency and effectiveness.

A critical issue with GAPI is that it suffers from considerable training time while instantly providing accurate recommendations. In the evaluation, we used a small dataset to train GAPI, however it takes very long time to finish the training, even using a powerful computer. In this respect, we anticipate that it might be impractical to train GAPI with real-world datasets.

The rationale behind the prolonged execution time is as follows. The underpinning graph neural network of GAPI relies on the use of nodes and edges to encode the relationship among API function calls. In particular, there are two main parameters, i.e., the kernel size and the node embedding size. In the experiment, the node embedding was set to 64, resulting in 14,366 nodes only for the smallest dataset considered in the evaluation, which consists of only 200 projects. In this respect, we see that even for a very small amount of source code, GAPI has to use a large number of parameters to learn from the data. This causes computational complexity in the whole approach, leading to prolonged execution time. This is further confirmed by our previous work, i.e., according to our experience gained from recent studies [68], graph neural networks are computationally expensive, especially when the representation of the input data in the graph format is not properly optimized. Our empirical evaluation shows that, by using a suitable data representation scheme, we dramatically decreased the number of nodes, and thus the execution time, while still obtaining a high prediction accuracy.

The recommendation engine of LUPE is built on top of an Encoder-Decoder neural network, and thus it is suitable for working with sequential data. To feed LUPE, we transform each method definition into a sequence of APIs. This encoding scheme allows the recommendation engine to properly learn the relationship between the APIs in the training phase. Eventually, this helps LUPE to obtain a good prediction performance in the testing phase.

Considering a longer list of recommended items, i.e., a large value of N, helps increase the recall values, but decreases the precision ones. Moreover, in fact a long list may tire the developers as they have to scan it to select the most suitable ones. Therefore, selecting a suitable value of N to maintain a trade-off between effectiveness and efficiency is an important issue, and we will investigate this in our future work.

4.2.5 Threats to validity

Threats to *construct validity* concern the relationship between theory and observation. In particular, they are related to how the simulated settings used to evaluate the systems reflect a real application scenario. Our evaluation approach attempts to simulate the presence of incomplete code. In a real usage scenario, the order in which one writes code may not resemble our simulation. Thus, a usage evaluation through controlled experiments with developers involved in a realistic programming task is planned as a part of our future agenda.

Threats to *internal validity* are the confounding factors internal to our study that might have an impact on the results. The comparison with the baselines might be subject to bias. We mitigate this threat by curating dedicated datasets being eligible as input for LUPE and the two baselines. The same experiment settings related to k-fold cross validation and have been imposed on the systems. Moreover, to compare with GAPI, we used the original implementation made available by its authors,[31] and ran the experiments with GAPI and LUPE using the same experimental settings. To further make sure that the evaluation is fair, we trained GAPI on the new dataset by tuning its hyperparameters, and selecting the ones achieving the best prediction performance.

Threats to *external validity* are the generalizability of our results. Such generalizability concerns *(i)* on the one hand, the recommenders on which the evaluation has been carried out (conclusions may or may not apply to other recommenders); and *(ii)* on the other hand the considered datasets.

[31] https://github.com/laurence-ling/GNNAPIRec

4.3 Conclusion

Software development activity has reached a high degree of complexity, guided by the heterogeneity of the components, data sources, and tasks. The proliferation of open-source software (OSS) repositories has stressed the need to reuse available software artifacts efficiently. To this aim, it is necessary to explore approaches to mine data from software repositories and leverage it to produce helpful recommendations. This chapter presented two approaches that address notable limitations in the domain by reusing existing techniques from other domains. FOCUS provide developers with suitable API function calls and code snippets while they are programming. A thorough evaluation has been conducted *(i)* on different OSS datasets to study the approach's performance, and *(ii)* in a user study with 16 participants to assess the perceived usefulness of FOCUS recommendations. We succeeded in integrating FOCUS into the Eclipse IDE, and we made available online the developed tool together with the parsed metadata [186]. This aims at providing the research community at large with a sound replication package, which then allows one to seamlessly reproduce the experiments presented in the chapter .

In the second part of the chapter, we presented LUPE as a novel approach to API recommendation, exploiting a machine learning translation technique. Through an empirical evaluation conducted on data from the Android domain, we showed that the proposed tool obtains a satisfying prediction performance on real-world datasets, thereby outperforming state-of-the-art baselines, GAPI [145] and FACER [9].

Chapter 5

Categorizing GitHub projects

Over the last decade, open-source software repositories have gained a prominent role in storing and managing software projects. Different kinds of artifacts, including source code, mailing lists, bug tracking systems, and documentation, are stored and managed homogeneously employing powerful technologies. In such a context, GitHub is playing the role of forefront platform managing more than 190M repositories, with more than 28M of them being available to the public.[1] The platform offers developers the possibility to classify the stored artifacts by means of topics, i.e., it introduced the possibility of labeling the stored repositories only in 2017 to help developers increase the reachability of their repositories. The assigned labels allow users to characterize projects, e.g., with respect to the provided functionalities and employed technologies.

However, assigning wrong labels to a given repository can compromise its popularity [37], and even worse, prevent developers from finding projects that they might be willing to contribute. In this respect, we come across the following motivating question:

> "*Which label should I use to annotate this new project managed by means of the employed OSS repository?*"

Repo-topix [84] is a mechanism based on information retrieval techniques, and it has been developed and integrated into GitHub to recommend topics. Unfortunately, neither the source code nor the experimental dataset was available at the time of writing, thus preventing us from reusing Repo-topix as the baseline.

In an attempt to improve this tool, e.g., in terms of the variety of the suggested topics, this chapter presents two automatic approaches to categorize GitHub repositories given the README content, i.e., the MNBN and HybridRec.

[1]The numbers are collected at the time of writing.

First, we apply the well-founded Multinomial Naïve Bayesian (MNB) network to extract README files' content, source code and eventually to recommend topics. This work has been published at the Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering [72], and the candidate contributes to the development and assessment of the whole MNBN approach. It is fed with a TF-IDF vectorization of README files, which represent the most frequent terms over all documents. As the final output, the tool retrieves the best-ranked topics according to their probabilities, and suggests them to developers. Due to the lack of a suitable baseline at the time of writing, we assess the quality of the results by means of suitable metrics for the examined domain. However, such a tool can recommend only *featured* topics, i.e., a set of topics, which are curated by GitHub. Subsequently, we successfully developed another approach, called TopFilter [66], that relies on a collaborative filtering technique to extend the recommendation capabilities of MNBN, taking into consideration non-featured topics.

In this respect, we show that the combined use of MNBN and TopFilter (named TopFilter^{+} hereafter) can improve the results obtained by MNBN. However, according to further investigations, there is still room for improvement for TopFilter^{+}. In particular, it is necessary to deal with unbalanced datasets (as typically occur in real contexts) as well as to enhance the quality of the recommended items.

Therefore, the second part of the chapter is dedicated to HybridRec, a hybrid recommender system which retrieves topics for repositories using a combination of stochastic and collaborative filtering recommendation strategies (thereby yielding the name *hybrid*), being capable of dealing with unbalanced and heterogeneous datasets.

The candidate contributes to the development of the stochastic component, as well as the corresponding evaluation. HybridRec has been published in the Applied Intelligence journal [70].

To enable both techniques we select the most used artifacts[2] by relying on popular topics. A crucial step lies in the preprocessing phase of the dataset and topics, which dramatically differs from the one set in place for the GitHub repositories. In particular, the type of available metadata for each project can affect the recommendation outcomes as well as the efficiency of the underpinning techniques. To evaluate HybridRec, we perform a series of experiments, exploiting real-world datasets collected from GitHub. Moreover, we compare HybridRec with MNBN [72], TopFilter, and TopFilter^{+} [66], which can be considered as among state-of-the-art approaches to the recommendation of topics for GitHub repositories.

[2]For the sake of the presentation, the terms "*project*," "*repository*," and "*artifact*" are used interchangeably throughout the chapter.

Outline of the chapter: The first part of the chapter presents the MNBN approach. First, Section 5.1.1 shows a motivating example and a set of challenges related to the categorization of GitHub repositories. The approach is overviewed in Section 5.1.2 and Section 5.1.3 describes the materials used in the evaluation. The outcomes are presented in Section 5.1.4 and we conclude the first part by discussing threats to validity in Section 5.1.5. Afterward, we introduce HybridRec in the second part of the chapter. The whole approach and its submodule are presented in Section 5.2.1. Section 5.2.2 reports the datasets and the metrics employed in the evaluation. The obtained results and the threats to validity are discussed in Section 5.2.4 and Section 5.2.5 respectively. Finally, we conclude the chapter in Section 5.3.

5.1 MNBN

5.1.1 Motivation and background

Software developers use open-source software (OSS) repositories to publish and disseminate their work. GitHub is amongst the most popular platforms that offer open environments where developers can share their code and interact with each other. To assist developers in approaching projects of their interest, GitHub provides users with tools and techniques that help narrow down the search scope and increase the visibility of its projects. In particular, to characterize the stored artifacts, GitHub leverages *featured topics*,[3] i.e., a list of the most popular and active topics, which are periodically monitored and updated. Such a public list indicates the community's trend in terms of the most used topics. Developers have manually assigned GitHub topics according to their experience as well as to the content of the repositories at hand. Nevertheless, topics assigned in such a way might be inaccurate or represent wrong concepts, thus compromising the visibilities of projects.

Fig. 5.1 represents an explanatory example consisting of the *longclaw* repository,[4] which implements extensions of Wagtail CMS[5] to enable the development of typical *e-commerce* functionalities. By looking at the list of topics, users can easily understand that the project employs *django*, a *python* web framework. Though the given topics characterize the repository's features, some of them might be considered redundant, e.g., *python3*, *python-2*, and *wagtail-cms*. Moreover, only three topics are considered to be *featured* as they are given in Table 5.1.

The example in Fig. 5.1 shows that on the one hand, user-specified topics are usually more extensive than the *featured* ones. On the other hand, two different topics can express

[3] https://github.com/topics
[4] https://github.com/JamesRamm/longclaw
[5] https://github.com/wagtail/wagtail

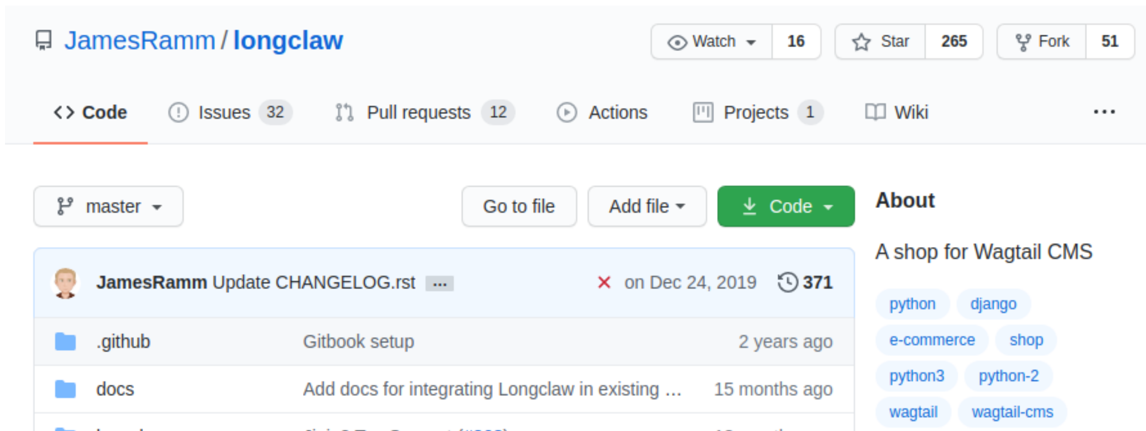


Fig. 5.1 The *longclaw* repository from GitHub with different topics.

Table 5.1 The *longclaw* repository topics.

Topics	GitHub featured topics
python, django, e-commerce, shop, python3, python-2, wagtail, wagtail-cms	python, django, wagtail

the same concept, and a larger set of topics could affect the prediction accuracy of automatic approaches that might rely on such data.

Summary. Topics are an effective means of summarizing the main features of a GitHub repository. Furthermore, the proper use of topics facilitates the searchability and discoverability of different items. Thus, there is the need for techniques and tools to automatically generate topics to prevent them from being misleadingly/wrongly established and not correctly reflecting the contents of the considered projects.

Challenges in mining the GitHub ecosystem According to existing work [55, 124], extracting valuable data from GitHub is a daunting task and exhibits several pitfalls. We identify the following challenges to be undertaken to conceive the desired recommender system:

C1: ***Data redundancy*.** GitHub topics are specified (or even created) by users to classify their repositories. However, this manual process results in inaccurate labeling in some situations. For instance, a user can define both *python* and *python-3* as topics for its repository, which are redundant. Refining the list of topics by removing such kinds of topics can improve the discoverability of the repository;

C2: ***Structure of available metadata*.** Concerning the available sources of knowledge, a standard GitHub project is described by one or more README file(s), a brief de-

scription, and possibly by a Wiki. Furthermore, there are different kinds of accessible metadata, e.g., commits, issues, forks, and stars. Though GitHub provides publicly access in most of the cases, extracting valuable information from the data mentioned above requires a set of tailored preprocessing techniques;

C3: ***Popularity mechanisms*.** GitHub provides users with the *star* and *forking* mechanisms to assess the popularity of a given repository [37, 121]. The former is used to improve the visibility of a project. The latter is typically employed when a developer wants to "*freely experiment with changes without affecting the original project*."[6] Moreover, GitHub groups the most popular projects under a curated list, i.e., featured topics. In such a way, the popularity of a repository helps the mining process filter out unuseful artifacts, e.g., toy and dummy projects. Though many software artifact repositories provide popularity mechanisms, there is no uniform way to define the popularity of an artifact. For instance, GitHub includes many elements, i.e., *star*, *forking*, number of committers, ifnextchar.etcetc., to assess the repository popularity, while Maven defines the popularity of an artifact by relying on the number of usages, i.e., when another project employs the artifact;

C4: ***Crawling and Data Dump*.** The availability of input data is a crucial aspect of the recommendation process. To gather them from a platform that stores OSS projects we can *(i)* use a crawler or *(ii)* rely on a data dump stored in some database format. Concerning the crawling activity, GitHub exposes its API to obtain all needed data by exploiting different libraries, e.g., JGit,[7] PyGitHub,[8] to name a few. Furthermore, GHTorrent [97] offers a regularly updated data dump of the entire platform in several formats, e.g., SQL, and MongoDB. It is worth noting that GHTorrent does not include the actual content of each repository, but only stores metadata;

C5: ***Configurations of the underpinning recommender*** *systems*. Depending on the features of the considered OSS ecosystem, the employed recommendation algorithm must be adapted accordingly by changing its internal configurations. Such a phase can rely on different parameters that vary according to the nature of the used algorithms.

[6] https://docs.github.com/en/free-pro-team@latest/github/getting-started-with-github/fork-a-repo
[7] https://www.eclipse.org/jgit/
[8] https://pygithub.readthedocs.io/en/latest/index.html

Research objective. Considering the need mentioned above and the corresponding challenges, we propose a workable solution to the recommendation of topics to GitHub repositories. We conceptualize a hybrid use of a complement naïve bayesian network and a collaborative-filtering technique.

5.1.2 The MNBN approach

Given a GitHub repository, our approach aims to suggest a set of relevant featured topics by analyzing its README file and its source code. A typical scenario involves a developer who looks for suggestions while she is working with the repository. Our approach helps increase the visibility of the repository on the platform by suggesting pertinent topics. Different from other GitHub project classifiers, we turn from a standard classification to a multi-classification problem by using different modules of the source code classifier (SCC) infrastructure of the *scikit-learn* library.[9] In particular, the MultinomialNB (MNB)[10] and TF-IDF vectorizer[11] components are used as follows: *(i)* we first create a curated dataset by selecting README files for each selected featured topic; *(ii)* then, the MNB is trained with vectors computed by the TF-IDF vectorizer; *(iii)* once we have the predicted list of topics, we discover the programming language with the GuessLang tool[12]; *(iv)* finally, we combine the results and deliver the list of recommended topics.

The architecture of the proposed approach is shown in Figure 5.2 and described in greater detail in the following subsections.

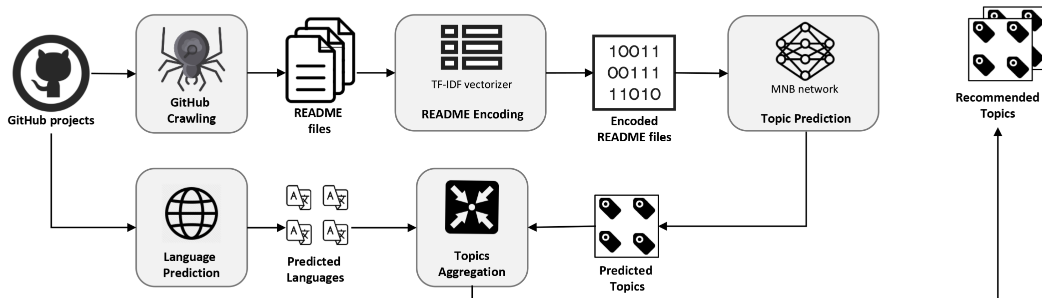


Fig. 5.2 Overview of the proposed approach.

[9] http://scikit-learn.org
[10] https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html
[11] https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html
[12] https://pypi.org/project/guesslang/

5.1.2.1 GitHub Crawling

To create a dataset with README files coming from GitHub projects, we used PyGithub [6], a library that provides a complete set of APIs to interact with GitHub repositories. Using a project's name as query, we can retrieve different types of information such as commits, issues, and name of the repository's owner. For each repository, we downloaded only its README file(s) to provide input to the MNB network. The rationale behind the selection is that README files provide an informative description of the project being developed,[13] e.g., among others, what the project does, or how developers can start with it [206]. In this sense, we assume that information coming from README files is suitable for the recommendation process.

Concerning the topics, we consider only the featured ones for two reasons: *(i)* it is not feasible to train a network for all possible topics available on the platform; and *(ii)* the featured topics are the most popular according to GitHub's statistics; this means that they can offer a pretty good coverage of the community's interests. Among these topics, we deployed an additional filter on the query to limit the number of topics considered in the training as this phase suffers from a decrease in performance with too many categories. By using the GitHub query language [7], we applied the following query filter:

$$Qf = \text{"}is : featured\ topic : t\ stars : 100..80000\ topics :>= 2\text{"} \tag{5.1}$$

to consider only GitHub repositories having a number of stars between 100 and 80,000, and tagged with at least two topics. We set such a filter after several query refinements, as the chosen featured topics have different degrees of popularity. For instance, the most starred project of the *3d* topic has around 53,000 stars; reversely, some other topics have top rank projects that do not reach 1,000 stars. Thus, we tried to select the best query filter to include a sufficient number of repository for each topic. By imposing such a filter, we tried to avoid skewed repositories that may not have an informative README, or projects that are already abandoned for a long time.

GitHub developers use *stars* as a voting mechanism to foster the popularity of a certain project [35]. Through this system, each user can support her/his favorite projects available on the platform. *Forking* is another index related to the quality of the project. This feature is typically employed as a starting point for a new project [121]. Furthermore, there is a strong correlation between forks and stars [37]. In this sense, we suppose that a project with a high number of stars means that it gets attention from the OSS community, and thus being suitable to identify popular repositories [38, 36].

[13] https://help.github.com/en/github/creating-cloning-and-archiving-repositories/about-readmes

5.1.2.2 README encoding

Before the training process, we encode the content of each README using the TF-IDF vectorizer provided by scikit-learn. The library embeds all the Natural Language Processing (NLP) techniques to preprocess README files, i.e., stop word removal, stemming and lemmatization. After the preprocessing phase, the vectorizer computes the TF-IDF weighting scheme to find the most representative terms over all documents. The TF-IDF computes the inverse document-frequency using the formula showed in Equation 5.4:

$$\text{idf}(t) = \log \frac{1+n}{1+\text{df}(t)} + 1 \tag{5.2}$$

where n is the total number of documents in the document set; *df(t)* is the number of documents in the document set that contain term t. The encoding is applied to the entire content of each README file to be fed to the MNB network. This is a preparatory phase to the training and it is conducted only at the beginning of the process.

5.1.2.3 Topic Prediction

Once we encoded the data using the mentioned techniques, we feed the model to perform the training phase. Naïve Bayesian network is a probabilistic model based on the Bayesian theorem that expresses the probability of a certain event given a set of preconditions [122]. The terms "Naïve" refers to the assumption that all the features are conditionally independent. This means that the classifier reaches a higher performance if each class does not have any relationship with the others. However, this condition does not always hold in practice and the model is used with relevant results. In our work, we use a Naïve Bayesian network based on a multinomial distribution, defined as follows:

$$P(y \mid x_1, \dots, x_n) \propto P(y) \prod_{i=1}^{n} P(x_i \mid y) \tag{5.3}$$

where $x_1, \dots, x_n$ are the features and y the class to be predicted with a certain probability P. We decided to employ the technique as it has demonstrated itself to outperform other ML models in the context of text classification [15]. To improve the performance of the network, we applied the TF-IDF scheme on the input data (see Section 5.1.2). Such a preprocessing phase should possibly enhance the quality of predicted items of the Bayesian classifier as this has been confirmed by an existing work [128].

As previously mentioned, we exploited the MNB implementation provided by scikit-learn. By default, MNB predicts only one class for each sample, we modified this feature by ranking

all the results to include the most probable topics as we aim to recommend more than one topic.

5.1.2.4 Language Prediction

Among featured topics, 20 of them are related to the main programming languages, e.g., Java, Python, C. Detecting them in a GitHub project requires a comprehensive analysis of the source code. Consequently, the MNB network is not suitable for this type of analysis due to its internal construction. However, since language is an important tag, we attempted to recommend it by employing GuessLang, a Python library which was tailored for this purpose. As it has been claimed in the documentation, GuessLang can predict 20 different programming languages with satisfying accuracy. This component is executed as stand-alone instance to predict the language for each analyzed repository. To correctly predict the programming language, this module analyzes all files within a repository. Due to the timing and memory constraints in the computation, we limit the number of analyzed files to 1,000 for each repository that does not exceed 10KB in size.

5.1.2.5 Topics Aggregation

The final step consists of combining the predicted topics coming from the MNB network and the language discovered by the tool. To aim for reliable results, we replace the last element of the predicted topic list with the programming language delivered by GuessLang. Let $T = t_1, t_2, ..., t_n$ be the list discovered by the MNB model, and t_l be the topic related to a programming language, then as the last element of T is the less probable of the retrieved set, the final output is the list $T_f = t_1, t_2, ..., t_l$, where t_l becomes the new last element.

5.1.3 Evaluation

This section presents the evaluation conducted to study our proposed approach. In particular, Section 5.1.3.1 presents the research questions we wanted to answer by means of the performed experiments and thus, to study the approach's performance. Section 5.1.3.2 introduces the datasets we populated to serve as input for the experiments.

5.1.3.1 Research questions

By performing the evaluation, we aim at addressing the following research questions:

- **RQ**$_1$: *How does the variation of training data impact on the prediction performance?* To answer this question, we varied the dimension of the dataset to include

more training data. In particular, we assess the quality of three different datasets to find out the best one in terms of success rate.

- **RQ$_2$**: *Is the approach able to provide consistent recommendations considering featured topics?* We compute the metrics to measure the relevance of our suggested topics considering the distribution of the considered repositories.

5.1.3.2 Dataset

To build the dataset, the quality filter discussed in Section 5.1.2 was exploited to select 134 different featured topics. The final aim is to build a balanced dataset in which every topic does not surpass the others in terms of influence, so as to avoid possible negative impacts on the final results.

The employed MNB network is slightly different from other models because it works on probability. Thus, we want to measure how the dimension of training data can affect the accuracy of the model. In this way, we created a global dataset of 13,400 README files by considering 100 repositories for each topic. Moreover, since we can recommend 20 additional topics related to the programming languages exploiting the GuessLang module, summing up our approach is able to predict a total number of 154 topics.

The number of topics for each repository has an important impact on the recommendation quality. As several repositories may have a low number of featured topics, it is necessary to examine the distribution of number of featured topics over the repositories. If we call c as the cut-off value for the number of topics, then Figure 5.3 displays such the distribution by varying c. The y-axis represents the percentage of repositories with respect to the created golden dataset, while the x-axis corresponds to c. We can see that, if we set the cut-off value $c = 1$, then 100% of the repositories have at least 1 featured topic. However, the percentage of repositories decreases considerably when c increases. For instance, only 40% of the repositories have c=3 featured topics and only 10% of them have c=5 featured topics. Given the circumstances, the prediction of more than 5 featured topics becomes a hard task considering this strict assumption.

5.1.3.3 Experimental settings

We performed ten-fold cross-fold validation [132] on all the datasets. In particular, the README files used in each dataset are divided into 10 equal parts. To preserve a balanced training set along with all the training phases, for each topic, we used 90% and 10% of the files for training and testing, respectively.

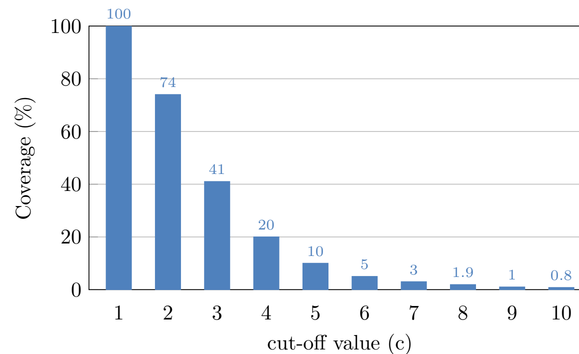


Fig. 5.3 Distribution of the number of topics in the dataset.

Figure 5.4 depicts the testing process for a single test file. In particular, we computed the TF-IDF vectors for each test README file to get predicted topics. Then, we retrieved the real topics from GitHub projects. Using the API provided by the PyGithub library, we mapped each repository to its real topics. From these, we filtered out the ones that are not featured topics because the network is not able to retrieve hand-made topics out of the train set. To aim for a better coverage, some lightweight NLP techniques have been applied to both the predicted topics and user topics, i.e., stemming and dashes removal. This processing phase has no impacts on the topics semantic, as the employed techniques work on syntactical aspects. For example, the topics *data-structures* and *datastructures* are semantically the same and we consider them as a true positive during the accuracy measurement. The last step involves the comparison of real topics and the recommended ones. To have a qualitative measure of the approach, we computed all the evaluation metrics for each test file and we are going to report the results in the next section.

5.1.4 Results

5.1.4.1 Explanatory example

Before going into detail, we take a concrete example to illustrate how the system recommends topics in Table 5.2. In this table, there are 10 repositories and each of them includes a number of real and featured topics. For each repository, we compare its featured topics with the top-5 results retrieved by our approach.

As it has been mentioned in Section 5.1.2, topics that are out of the training set from the real topics of a repository are removed. The last proposed topic is discovered by the language predictor component and the matched topics are printed in bold to distinguish them with the others. By all repositories, there is at least one matched item, i.e., success rate is equal to 1. Among others, given the testing project fishercoder1534/Leetcode, all the recommended topics are matched with those from the ground-truth data.

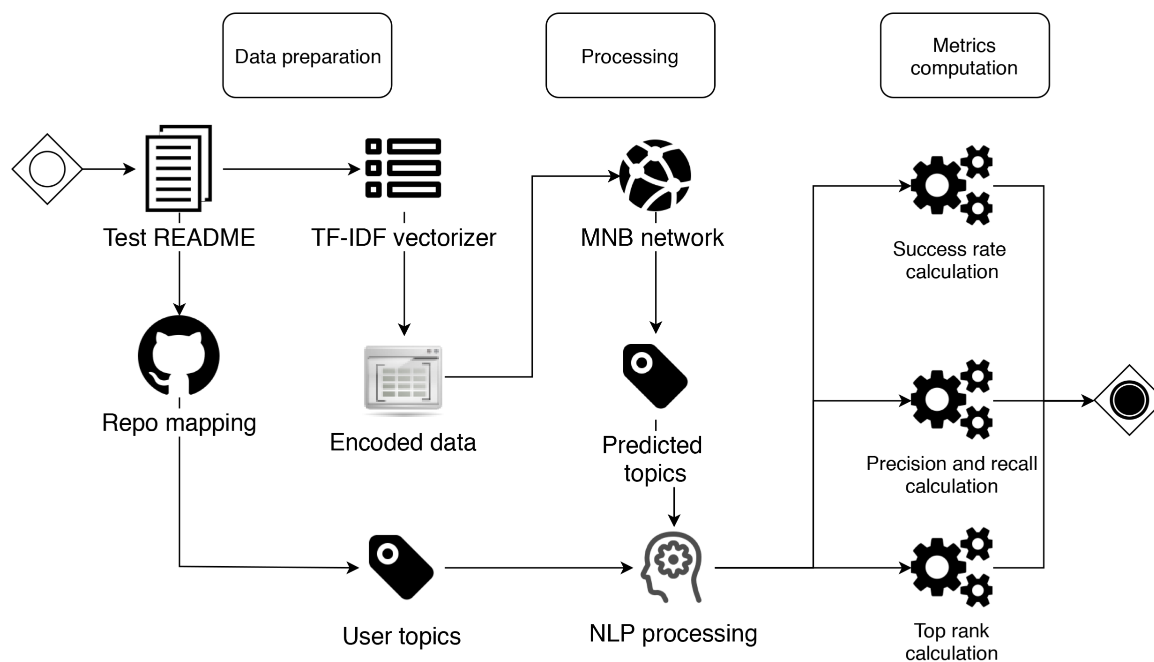


Fig. 5.4 The testing phase for a single project.

As it can be seen, although the other recommended items do not belong to the ground-truth data, they are well correlated with the suggested ones. To be concrete, the topic *linux* is strongly related to *ubuntu* in the esell/deb-simple project although it does not appear in the real topics set. Given the circumstances, we suppose that the application of a lexical database to map words with similar semantic meaning such as WordNet [163] should possibly increase the overall prediction performance.

5.1.4.2 Result analysis

RQ$_1$: *How does the variation of training data impact on the prediction performance?* Starting from the dataset described in Section 5.1.3 three different datasets have been populated, i.e., D_1, D_2, and D_3 by incrementally enlarging the considered training files as described in Table 5.3. We applied the same experimental settings for all the datasets. Dataset D_1 consists of 1,340 README files and 10% of each topic are used as testing. Dataset D_2 and D_3 follow the same structure in which we increase the number of README files up to 50 and 100, respectively.

By running the tool on the datasets, for each testing item, we obtained a ranked list of topics that is considered to be relevant. Given the multi-classification problem, the number of recommended items dramatically affects the final results. As it has been shown in Figure 5.3, generally the repositories contain a low number of topics. Thus, we varied the number

Table 5.2 Example of repositories, their topics and the recommended topics.

Repository	Real topics	Featured topics	Recommended topics
a1studmuffin/SpaceshipGenerator	python,blender-scripts, spaceship, procedural-generation, game-development, 3d	python, 3d	shell, terminal, **3d**, opengl, **python**
0xAX/go-algorithms	golang, algorithm, data-structures, go, sort, tree-structure	algorithm, data-structures, go	**data-structures**, **algorithm**, twitter, library, **go**
ajenti/ajenti	ajenti, python, javascript, administration, linux, panel, angular	python, javascript, linux, angular	firefox, webpack, **linux**, **angular**, **python**
fishercoder1534/Leetcode	leetcode, algorithm, java, interview, mysql, bash, apache, data-structures, leetcode-solutions, leetcode-questions, leetcode-java, leetcoder	algorithm, java, mysql, bash, data-structures	**algorithm**, **mysql**, **bash**, **data-structures**, **java**
cryogen-project/cryogen	clojure, cryogen, static-site-generator	clojure	jekyll, **clojure**, atom, gulp, html
alizahid/slinky	jquery, navigation, mobile, es6, javascript, menu, plugin, yarn, babel, webpack, css, sass	jquery, mobile, javascript, es6, css, sass, babel, webpack	**es6**, **jquery**, **css**, **sass**, **javascript**
esell/deb-simple	maintainer-wanted, golang, go, debian, ubuntu	go, ubuntu	**ubuntu**, ansible, linux, shell, **go**
JamesRamm/longclaw	python, django, e-commerce, shop, python3, python-2, wagtail, wagtail-cms	python, django, wagtail	**wagtail**, **django**, serverless, express, **python**
nitin42/terminal-in-react	terminal, react, design, javascript, svg, webpack, webapp, css	terminal, react, javascript, webpack, css	**webpack**, eslint, react-native, **react** , **javascript**
kerl/kerl	erlang, otp-release, kerl, homebrew, shell	erlang, homebrew, shell	elixir, **homebrew**, bash, deployment, **shell**

Table 5.3 Datasets.

Dataset	# of testing files	# of training files	Exe. time (sec)
D_1	134	1,206	658
D_2	670	6,030	3,782
D_3	1,340	12,060	7,823

of retrieved topics for each testing item with a small value, i.e., $c = 2$, $c = 5$, and $c = 8$ recommended topics to find out the best configuration with respect to the returned items.

If N is the number of correctly predicted topics then for each setting, the corresponding quality metrics are computed with respect to N. The final success rates are depicted in Figure 5.5, Figure 5.6, and Figure 5.7 for $c = 2$, $c = 5$ and $c = 8$, respectively.

In Figure 5.5, it is evident that a better success rate is obtained when N=1, and this holds for all the datasets, i.e., D_1, D_2, and D_3. Running the tool on D_2 and D_3 yields a comparable success rate, i.e., 75.8% and 75.68%. This implies that at a certain point, increasing the amount of training data does not bring a radical change in the overall performance. This is interesting since it suggests that in practice, we just need a certain amount of training data

in order to get a decent accuracy. When N=2, it can be seen that the obtained success rate improves considerably along the addition of more training data. For example, by D_1 we get 23.1% as success rate, however the corresponding value by D_3 is 63.14%, which is almost triple larger than that of D_1.

By considering Figure 5.6 and Figure 5.7 together to study the approach's performance for the cases c=5 and c=8, we witness the same trend as that by c=2. To be concrete, requesting more of correctly predicted topics means getting a lower success rate. For example when c=5, by Dataset D_1 the maximum success rate is 80.15% and it is obtained when N=1. This score decreases dramatically for a larger value of N, i.e., with N=4 corresponding the success rate is 19.83% which is four times lower than that of the case where N=1. This suggests that obtaining a higher number of correct topics becomes more difficult. Similarly, by D_2 and D_3, success rate deteriorates quickly if we want to see more matched topics. By comparing the results yielded by all three datasets, we see that running the MNB network on D_2 brings a slightly better prediction performance in terms of success rate, and this is consistent with what we got for c=2. This is further confirmed in Figure 5.7 as using D_2 for training data yields a better success rate.

These findings should be explained by the internal construction of MNB. As the network uses probabilities without considering any other type of relationships among the features, it works well even with a small amount of data. Thus, the addition of further data has a negative impact on the learning rate of the MNB network. Concerning the number of recommended topics, our experimental configurations demonstrate that suggesting more topics increase the success rate. As a large part of the dataset contains repositories with at most 8 featured topics, those that have more than this number of topics are not statistically significant for our purposes. Additionally, we normalized success rate measures to be compliant with the distribution of the dataset.

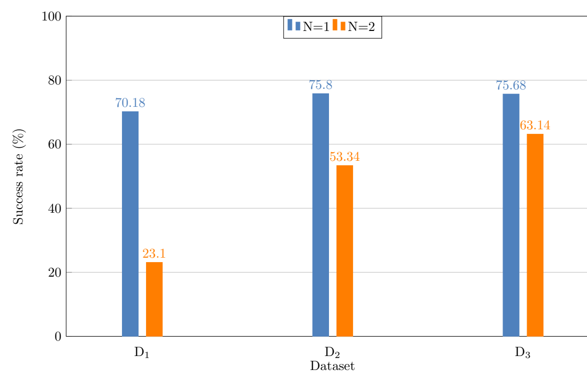


Fig. 5.5 Success rate for c=2.

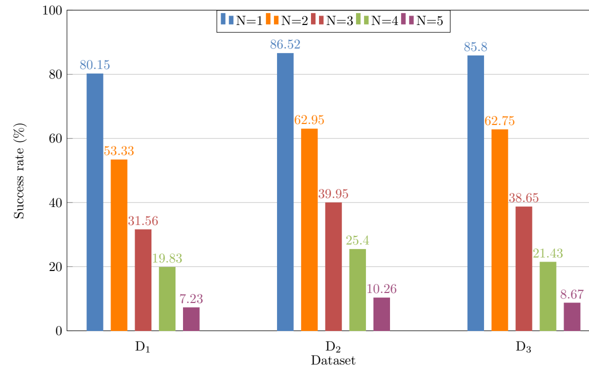


Fig. 5.6 Success rate for c=5.

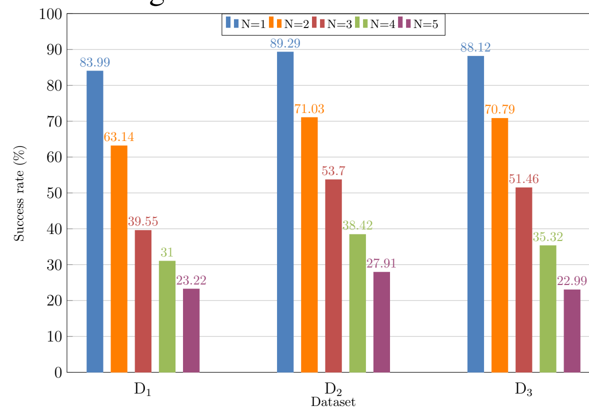


Fig. 5.7 Success rate for c=8.

Answer to RQ$_1$. Once a certain threshold has been reached, augmenting the training data does not bring a radical change in performance. In particular, adding more data to Dataset D_2 to yield Dataset D_3 has a negligible impact on success rate, precision, and recall.

RQ$_2$: *Is the approach able to provide consistent recommendations considering featured topics?* Considering once again the three experimental datasets shown in Section 5.1.3, we computed precision and recall by varying the two values c and N. The precision, recall, and top-rank scores for all the test configurations are shown in Table 5.4. As it can be seen there, precision and recall scores decrease when N increases, which is consistent with success rate. Again, using D_2 as input data helps us achieve a better precision and recall compared to D_1 and D_3. Using D_3 helps obtain a better result with c=2. However, the difference between the two recall values is negligible. As expected considering the previous results, Dataset D_1 contributes to the worst performance along with all the possible configurations of N and c.

These results can be explained by the particular nature of the user's topics. As shown in Table 5.2, the featured topics belonging to the real repositories are very few. The side-effect of this assumption has a particular impact on the precision value, that goes down when we increase the number of recommended items. Meanwhile, recall does not suffer from the

Table 5.4 Precision, recall, and top-rank.

	Precision			Recall			Top rank		
N	D_1	D_2	D_3	D_1	D_2	D_3	D_1	D_2	D_3
2	43.61	**47.98**	**47.98**	38.10	42.62	**42.96**	60.59	65.19	64.81
5	27.51	**31.20**	30.70	55.68	**63.45**	63.06	—	—	—
8	19.63	**22.05**	21.57	61.96	**69.77**	69.04	—	—	—

dataset composition, as this metric highlights the true positive ratio. We assume that the obtained results are strongly affected by the distribution of the user topics.

To further study the approach's performance, we computed Top Rank scores following Equation 3.5 and the results are depicted in Table 5.4. We considered only the first top rank items since all repositories belonging to the golden dataset have at least one featured topic. Thus, we exploit this index to assess the capability of the MNB network in a more reliable way. Once again, using D_2 brings a better top rank compared to using the other datasets. In particular, the Top Rank index increases to 65.19% from 60.59% when we run the tool on D_3 instead of D_1. Moreover, running the tool on D_3 places a burden on the overall performance. This confirms the findings of **RQ**$_2$, in which D_2 facilitates a better prediction.

Answer to RQ$_2$. Our approach is able to provide relevant results in terms of Top Rank scores given a decent amount of training data.

5.1.5 Threats to validity

We identify the threats that may adversely affect the validity of the evaluation, and the countermeasures taken to minimize them.

Threats to *internal validity* concern the criteria behind the selection of GitHub topics. As we already mentioned before, a GitHub's user can manually insert several numbers of topics. As this affects the quality of the final results, we considered only a limited set of featured topics. Another issue is related to the training phase that employs only README files to discover topics: They contain usually rough data that might not properly represent the entire repository's content. We mitigate this threat by obtaining a stable number of README files for each topic, and adding the GuessLang module to cover programming languages.

Concerning the threats to *external validity*, there might be issues with the selected dataset. First, we downloaded the projects directly from GitHub using the Python API. The platform limits the number of results to 1,000 for each query, so we were not able to access to the entire knowledge of the platform. Moreover, the query changes the set of the retrieved results at each run. Thus, the same query could retrieve different records into different runs. To avoid this situation, we imposed the above mentioned quality filter to cover projects as much as possible, without needing to execute a query several times.

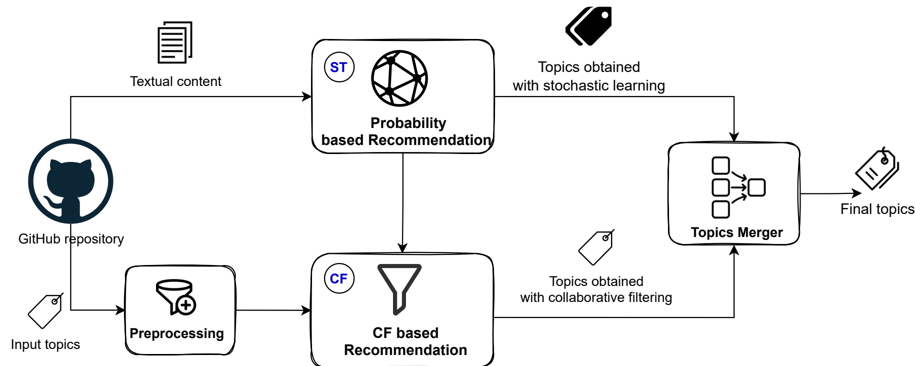


Fig. 5.8 Overview of the HybridRec approach.

Construct validity pertains to the conducted testing phase. As said, 10% of the README files for each topic are removed in every testing fold. In this way, we cannot test all possible permutation that might affect the training phase, as it requires the storage of a huge amount of data. To the best of our knowledge, the selected testing configuration is suitable to assess the approach's accuracy. Another issue concerns the chosen metrics to evaluate the approach: the overall precision considerably decreases due to the aleatory number of user's topics. We tackle this issue by setting the *Top rank* index to highlight the precision in terms of the first result.

5.2 HybridRec

5.2.1 HybridRec architecture

In this section, we present in detail HybridRec, the proposed recommender system to provide topics for GitHub repositories. HybridRec parses textual contents, e.g., README files, Wiki contents, and commit messages collected from GitHub, and employs a Complement Naïve Bayesian Network [212], or CNBN for short, to extract preliminary topics. The predicted topics are then fed as input to retrieve additional relevant topics by means of a collaborative-filtering technique. The outcome of this phase is combined with the preliminary topics to yield the final recommendations.

Figure 5.8 illustrates the architecture of HybridRec, which consists of two main components, namely *(i)* **ST**: Probability based recommendation using a stochastic network [212]; and *(ii)* **CF**: Collaborative-filtering based recommendation [182, 235]. The following subsections describe these components in greater detail.

5.2.1.1 ST: Stochastic-based Component

The architecture of the first component is depicted in Fig. 5.9. Data is crawled from GitHub and then vectorized by the *TF-IDF encoding* component. The obtained data is used to feed the CNBN component, which returns a set of most probable topics. By resembling the choice made in the original work [72], we consider only the *featured* topics, i.e., the most popular according to GitHub's statistics. In such a way, we can train CNBN with the projects labeled with featured topics that are representative in terms of coverage. Furthermore, various NLP techniques are applied to both predicted and actual topics to improve the quality of the outcomes, as we explain as follows.

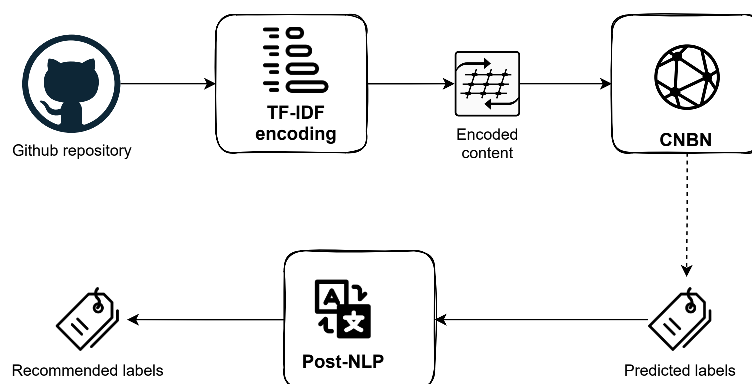


Fig. 5.9 The **ST** component.

TF-IDF encoding Given a repository, this module vectorizes the textual content of its artifact descriptions, using the *scikit-learn* library[14] that provides all the functionalities to encode textual content by finding the most representative terms. `Encoder` computes the inverse document-frequency using the following formula:

$$idf(t) = \log \frac{1+n}{1+df(t)} + 1 \quad (5.4)$$

where n is the total number of documents in the document set; *df(t)* is the number of documents in the document set that contain term t. A previous study [128] showed that applying such a weighting scheme should possibly enhance the quality of predicted items of the Bayesian classifier. Thus, we decide to apply this additional preprocessing step to improve the quality of the recommended topics.

[14] https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html

CNBN-based prediction In our previous work [72], a Multinomial Naïve Bayesian network (MNBN) has been used to extract topics, and the results were encouraging. In an attempt to improve the recommendation capability as well as to deal better with unbalanced datasets, we adopt an enhanced version of the Multinomial Naïve Bayesian Network called Complement Naïve Bayesian Network to compute the predictions. The term "*naïve*" refers to the assumption that all the features are conditionally independent. In other words, the network augments its prediction capabilities when each element has weak or no ties with the others. This model has been successfully employed to classify multinomial distributed data. Though MNBN and CNBN are similar from the structural point of view, their underpinning mechanisms used to compute the outcomes are completely different. In particular, MNBN employs a stochastic model that predicts a certain class **c** by relying on its training data. In contrast, CNBN makes use of the training data coming from all classes except **c**. In such a way, the performed estimation is more effective as the model uses a more even amount of training data per class. To be concrete, CNBN computes predictions by means of the following formula.[15]

$$\begin{aligned} \hat{\theta}_{ci} &= \frac{\alpha_i + \sum_{j:y_j \neq c} d_{ij}}{\alpha + \sum_{j:y_j \neq c} \sum_k d_{kj}} \\ w_{ci} &= \log \hat{\theta}_{ci} \\ w_{ci} &= \frac{w_{ci}}{\sum_j |w_{cj}|} \end{aligned} \tag{5.5}$$

where the summations are over all documents j not in class **c**, d_{ij} is the tf-idf value of term i in document j and a_i is a smoothing parameter used to compute the final weight w_{ci}.

This mechanism impacts positively on the computation of the weights by using the same input data. In fact, using CNBN instead of MNBN leads to better accuracy considering unbalanced datasets [212]. After this phase, CNBN produces a list of *top-N* topics according to their probability.

Post Natural Language Processing

The following *lightweight post-processing* steps are performed on the obtained topics, with the aim of further enhancing the prediction capabilities:

- **Dash removal**: All the dash occurrences in each topics are removed. For instance, the *build-system* topics becomes *buildsystem*. In this way, we enlarge the possible set of recommended items;

[15] We made use of the Python implementation of CNBN embedded in the ***scikit-learn*** library (https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.ComplementNB.html).

- **Stemming**: Using Porter's stemmer implementation provided by the *nltk* language processing library,[16] we squeeze each term to its root by cutting the suffix. Thus, terms such as *monitoring* or *testing* collapse to *monitor* and *test*, respectively.

5.2.1.2 CF: Collaborative filtering-based component

Though CNBN can recommend relevant topics by using textual contents, it provides only *featured* topics, i.e., a set of topics curated by GitHub. Therefore, it is necessary to have machinery to provide also non-featured topics. Being inspired by the TopFilter system [66], in this work we further extend the recommendation capabilities of the **ST** component to non-featured topics, using a collaborative-filtering recommendation engine.

Figure 5.10 provides an overview of the **CF** component. The `Preprocessing` module is used to obtain a *filtered* dataset. Given an initial set of topics already assigned to the repository of interest, `Data Encoder` encodes it in a graph-based structure to represent the mutual relationships between repositories and topics. From this, a project-topic matrix is created by following the typical user-item structure used in existing collaborative filtering applications [235]. Then, `Similarity Calculator` computes similarities among all the considered artifacts. Finally, the `Recommendation Engine` module retrieves a ranked list of top-N topics that are suggested by using user-based collaborative filtering technique [296]. The functionalities and preprocessing techniques implemented in **CF** are described in detail as follows.

5.2.1.3 TopFilter preprocessing

We adapt semantic mapping rules proposed in a recent work [120], revise them as well as propose our own rules, with the aim of enhancing the quality of the retrieved topics. These rules have been manually defined by building direct relationships among the terms. For instance, topics *firefox* and *chrome* are rooted to a common term, i.e., *browser*. In particular, we adopt two types of rules, i.e., Aggregating Rewriting Rules (ARR)s and Extending Rewriting Rules (ERR)s. The former are used to restrict the dictionary of possible topics by rewriting the topics in a different manner, e.g., `exporter` and `exporting` are aggregated in `export` that preserves their semantic. Meanwhile, the latter provide additional information about the topic, e.g., the repository containing the `sysmodule` topic are augmented by adding the `system` and `module` topics. Given a repository, ERRs extend the list of topic by adding new ones. Specifically, we perform the following ordered preprocessing steps:

[16] https://www.nltk.org/api/nltk.stem.html

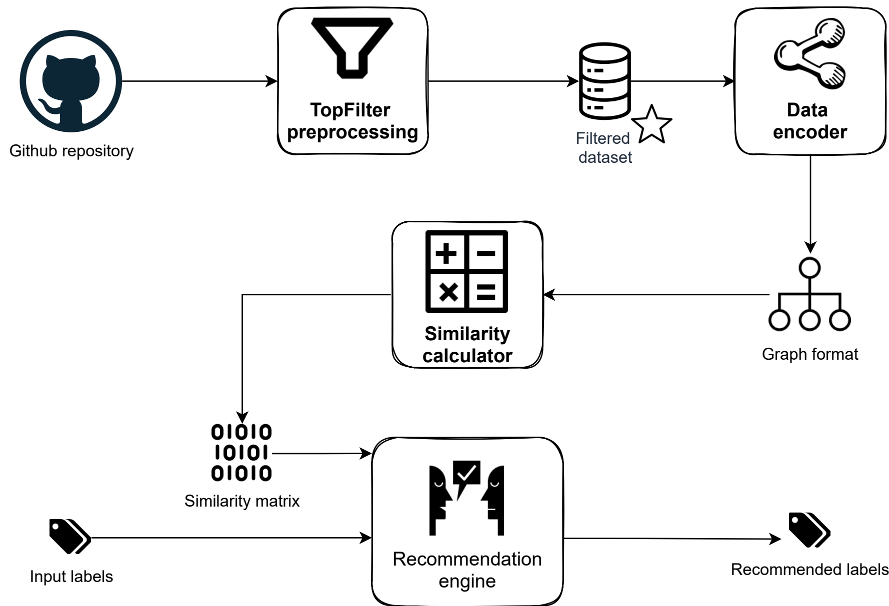


Fig. 5.10 The **CF** component.

1. **[ARR - *version removal*]** All the substrings referring to versions are removed from the topics. We use the following regular expression $v[\backslash d \backslash .]+$ to match versions. Then, the identified string matches are removed from the topics. For instance, this rule allows us to aggregate terms like `riot-api-v4` and `riot-api-v3` to the common term `riot-api-`;

2. **[ARR - *extra digits removal*]** Punctuation digits, non-English and non-ASCII characters at the end of the topics are filtered out. This rule enables us to match topics that are very similar apart from non alphabetic chars. For instance, using this rule will boil `python3`, `python2` and `python` down to the final term `python`;

3. **[ERR - *abbreviation expansion*]** Popular software engineering and computer science related abbreviations and acronyms, e.g., lib, config, DB, doc, are rewritten with their original form, e.g., library, configuration, database, document.

4. **[ERR - *split frequent topics*]** By computing the frequency of tokens, we split those that are made from the most frequent topics. For example, `javascript-tutorial` is split into two separate terms, i.e., `javascript` and `tutorial`;

5. **[ERR - *alias substitution*]** Relying on the topic aliases recently proposed [120], we transform topics according the identified aliases. As an example, `angularjs` is converted to `angular` following this rule;

6. **[ERR - *split tokens*]** Tokens are split based on the snake_case (terms separated by underscores), camelCase (terms separated with a single capitalized letters) or kebab-case (terms separated by hyphens) naming conventions. For instance, `springDemo` is split into `spring` and `demo`;

7. **[ARR - *nlp process*]** Stop words are removed, then NLP stemming and lemmatization methods are applied on the topics. For instance, `programming-in-haskell` becomes `program` and `haskell` topics;

8. **[ARR - *infrequent topics removal*]** Finally, topics with a frequency of less than a given threshold are removed.

5.2.1.4 Data encoder

This component represents the relationships between projects and topics in a matrix, whose rows and columns represent all the projects and all the corresponding topics. Thus, the cell (i, j) is set to 1 if the artifact in the ith row is labeled with the topic in the jth column, 0 otherwise. The matrix is eventually constructed by preprocessing raw topics with the same NLP techniques used in MNBN to filter out possible biased terms (see Section 5.1.2).

To illustrate how `Data Encoder` works, we consider a set of four GitHub repositories $P = \{p_1, p_2, p_3, p_4\}$ together with a set of topics $T = \{t_1 = junit$; $t_2 = testing$; $t_3 = specs$; $t_4 = module$; $t_5 = mocking$, $t_6 = mock\}$. Moreover, the *repositories-topics* inclusion relationships is denoted as $\ni$. A parsing of the projects reveals the following inclusions: $p_1 \ni t_1, t_2, t_6$; $p_2 \ni t_1, t_3$; $p_3 \ni t_1, t_3, t_4, t_5$; $p_4 \ni t_1, t_2, t_4, t_5$. After the NLP normalization steps, t_5 and t_6 collapse on the same term which is named as t_6. The final project-topic matrix is shown in Table 5.5.

Table 5.5 The *artifact-topic matrix* for the example.

	t_1	t_2	t_3	t_4	t_5
p_1	1	1	0	1	1
p_2	0	1	0	1	1
p_3	0	1	1	0	0
p_4	0	1	0	0	1

5.2.1.5 Similarity calculator

Given the encoded data, this module computes the similarity among the projects by considering the mutual relationships. Two nodes in a graph are considered to be similar if they

share the same neighbours by considering their edges. We represent a set of projects and their labels in a graph, so as to calculate the similarities among the projects. For instance, Fig. 5.11 depicts the graph-based representation of the project-topic matrix in Table 5.5.

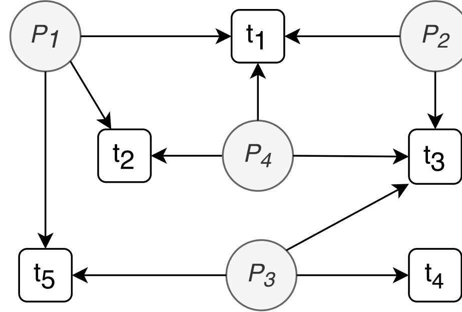


Fig. 5.11 Graph representation for repositories and topics.

Considering a project p that has a set of neighbor nodes (t_1, t_2,.., t_l), the features of p are represented by a vector $\phi = (\phi_1, \phi_2, .., \phi_l)$, with ϕ_i being the weight of node t_i computed as the *term-frequency inverse document frequency* function as follows: $\phi_i = f_{t_i} \times log(|P| \times a_{t_i}^{-1})$, where f_{t_i} is the number of occurrences of t_i with respect to p, it can be either 0 and 1. $|P|$ is the total number of considered projects; a_{t_i} is the number of projects connecting to t_i via corresponding edges.

Intuitively, the similarity between two projects p and q with their corresponding feature vectors $\phi = \{\phi_i\}_{i=1,..,l}$ and $\omega = \{\omega_j\}_{j=1,..,m}$ is computed as the cosine of the angle between the two vectors as given below.

$$sim(p,q) = \frac{\sum_{t=1}^{n} \phi_t \times \omega_t}{\sqrt{\sum_{t=1}^{n}(\phi_t)^2} \times \sqrt{\sum_{t=1}^{n}(\omega_t)^2}} \tag{5.6}$$

where n is the cardinality of the union of topics by p and q.

5.2.1.6 Recommendation engine

Given an input project p, and an initial set of related topics decided by the developer, the inclusion of additional topics can be predicted from the projects that are similar to p. The collaborative-filtering recommender works based on the assumption that *"if projects share some topics, then they will probably share additional topics."* In other words, it predicts topics' presence by means of those collected from the *top-k* similar projects using the following formula [182]:

$$r_{p,t} = \overline{r_p} + \frac{\sum_{q \in topsim(p)} (r_{q,t} - \overline{r_q}) \cdot sim(p,q)}{\sum_{q \in topsim(p)} sim(p,q)} \tag{5.7}$$

where $\overline{r_p}$ and $\overline{r_q}$ are the mean of the ratings of p and q, respectively; q belongs to the set of top-k most similar projects to p, denoted as $topsim(p)$; $sim(p,q)$ is the similarity between the active project and a similar project q, and it is computed using Equation 6.2.

The next sections present the experiments performed to evaluate HybridRec as well as to compare it with MNBN, TopFilter and TopFilter$^+$.

5.2.2 Evaluation

This section describes the process conducted to study the performance of HybridRec. In particular, three research questions are presented in Section 5.2.2.1 to address different aspects of the quantitative and qualitative evaluations. Afterward, an informative description of the datasets exploited in the evaluation is given in Section 5.2.2.2. Section 5.2.3 eventually describe the evaluation metrics and process.

To facilitate future research, we made available the HybridRec tool together with the related data in GitHub.[17]

5.2.2.1 Research questions

We study the performance of our proposed approach by answering the following research questions:

- **RQ$_1$**: *How does HybridRec perform compared to TopFilter$^+$?* In our previous work [66], TopFilter$^+$ was used to predict topics for GitHub repositories, obtaining promising results. HybridRec has been conceptualized following the same line of reasoning and implementing various boosting mechanisms. This research question investigates the impact of the enhancement on the overall prediction performance by comparing HybridRec with TopFilter$^+$;

- **RQ$_2$**: *In comparison to MNBN, does CNBN contribute to a better HybridRec performance?* We compare the recommendation capability of the two considered stochastic networks, i.e., MNBN and CNBN, using an unbalanced dataset. This aims to find out the factors that contribute to gain in the prediction performance;

- **RQ$_3$**: *How do the preprocessing steps impact on the HybridRec performance?* We investigate the impact of the preprocessing steps on the collaborative-filtering component by experimenting with both preprocessed and raw datasets;

[17] https://github.com/MDEGroup/HybridRec

- **RQ**$_4$: *What are the key differences between GitHub and MVN Repository, and how do they impact on the whole HybridRec recommendation process?* GitHub projects available on MVN Repository[18] are characterized by different features as well as metadata; we investigate to which extent such varieties could affect the mining process and the prediction capabilities of HybridRec.

5.2.2.2 Data extraction

To answer the four research questions, we curate four different datasets, namely D_1, D_3, D_2, and D_M as shown in Table 5.6 and described as follows.

- D_1 is the original dataset already used in our previous work [72, 66] consisting of 11,694 GitHub repositories with 19,337 topics;

- D_2: As discussed in our previous work [66], infrequent topics negatively affect the prediction outcomes. In this way, we removed infrequent elements from the dataset to analyze their impacts on the overall recommendation phase. We firstly filtered the initial set of topics using their frequencies counted on the entire GitHub dataset. Afterward, topics that occur in less than 20 repositories were removed, obtaining D_2 with 6,253 repositories and 455 topics;

- D_3: To evaluate the impact of the preprocessing steps, we created the third dataset by applying the preprocessing rules presented in Section 5.1.2. The resulting D_3 dataset consists of 5,620 repositories labeled with 6,442 topics.

- D_M: To evaluate the performance of HybridRec on a different repository of artifacts, we collected a set of 2,932 unique projects from MVN Repository Repository using Beautiful Soup,[19] a well-founded Python scraping library. Tags in MVN Repository are well-maintained as they are not freely assigned by developers but by a central authority once artifacts have been made available on the platform. The D_M dataset contains the most popular tags belonging to the *top categories* curated list.

5.2.3 Evaluation process

We use the *ten-fold cross-validation* technique [132] to analyze the performance of our proposed approach. Fig. 6.8 depicts the evaluation process consisting of three consecutive

[18] https://mvnrepository.com/
[19] https://www.crummy.com/software/BeautifulSoup/

Table 5.6 Datasets features.

	D_1	D_2	D_3	D_M
# of artifacts	11,694	6,253	5,620	2,932
# of topics	19,337	455	6,442	489
Avg. number of topics	8.24	6.70	8.60	3.23
Avg. frequency of topics	16.13	42.10	29.90	36.5

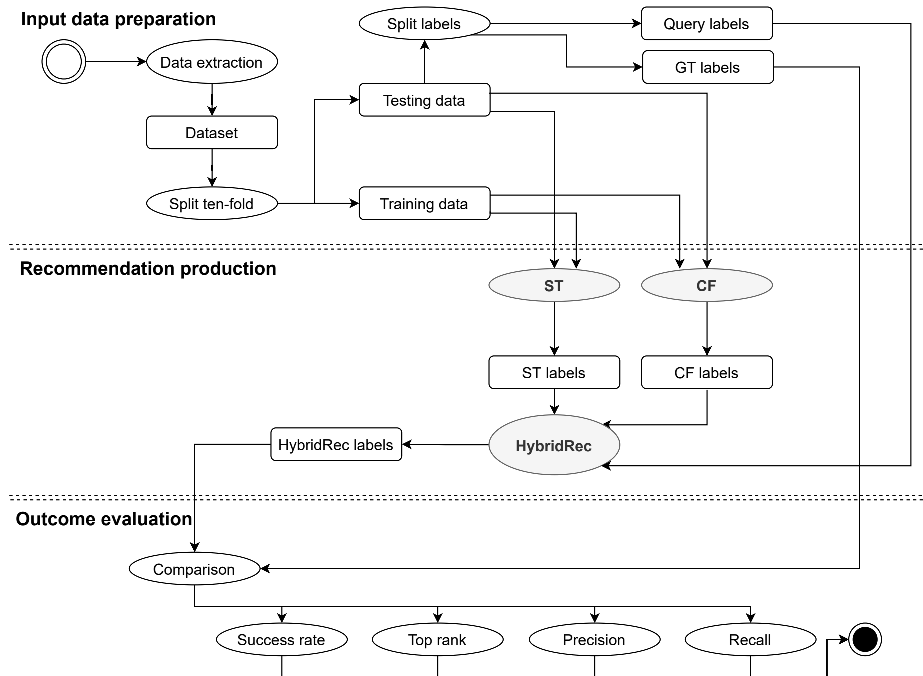


Fig. 5.12 Evaluation process.

steps, i.e., *Data Preparation*, *Recommendation Production*, and *Outcome Evaluation*, which are explained as follows.

Data preparation. This phase is conducted to collect repositories from GitHub that match the requirements defined in previous section during the `Data collection` step. The dataset is then split into a training and a testing set using `Split ten-fold`. Since the recommender systems are different in terms of input data, i.e., CNBN requires *README* files as input and training data, whilst HybridRec uses a set of assigned topics as input and for training, testing and training sets need to be specifically parsed for each individual approach. The `Split labels` activity simulates a real development scenario when a developer has already assigned some topics to her repository, and waits for recommendations, i.e., `Query labels`. For instance, given a GitHub repository and its topics, we fed the CNBN component using the repository README file annotated with the corresponding topics. Meanwhile, the testing data represent unlabeled README files that need to be categorized by the network. Contrariwise, TopFilter requires only training topics and an initial set of labels extracted

from the input repository, namely `Query labels` since the underpinning engine makes use of a collaborative filtering technique that suffers from the cold start problem.

Recommendation production. To enable the evaluation of HybridRec, we extracted a part of the topics for each testing project, resulting in the ground-truth data. The remaining part is used as a query to produce recommendations. We parsed and encoded text files in vectors using the TF-IDF weighting scheme to provide input to CNBN.

Outcome Evaluation. We evaluate HybridRec and compare it with MNBN, TopFilter, and TopFilter$^+$, analyzing the recommendation results by comparing them with those stored as ground-truth data to compute the quality metrics, i.e., *Success rate*, *Precision*, *Recall*, *Top rank*, and *Catalog coverage*.

5.2.4 Results

In this section, we study the performance of our proposed approach by and compares it with MNBN by answering the research questions presented Section 5.2.2.1. First, in Section 5.2.4.1 takes a concrete example to illustrate how TopFilter$^+$ and HybridRec recommend topics to a repository. Afterward, Section 5.2.4.1 and Section 5.2.4.1 report and analyze the experimental results.

5.2.4.1 Explanatory example

Before answering the research questions, we illustrate the recommendations provided by TopFilter$^+$ and HybridRec through a running example. As shown in Table 5.7, the maintainers of the repository *markdown-viewer*[20] labeled it with nine labels as listed on the *Original topics* column. The column *Processed topics*, shows the outcomes obtained from the preprocessing step on the original topics. The last two columns report the ranked recommendations provided by TopFilter$^+$ and HybridRec. Moreover, for each topic, Table 5.7 lists the corresponding frequencies over the D_2 and D_3 datasets. For instance, the `split token` rule (see Section 5.2.1) rewrites *chrome-extension* as *chrome* and extension, while the `alias substitution` rule generates *browser* from *chrome* and *firefox*. In other words, the rewriting rules refine the mined topics to generate more exhaustive and coherent ones. For instance, it is evident that *firefox-extension*, *chrome-extension*, and *browser-extension* can be better represented by their corresponding short forms: *browser*, *firefox*, *chrome* and *extension*. By looking to the original topics and the preprocessed ones, we can see that the semantics of the topics are preserved but their frequencies are increased. TopFilter$^+$ and HybridRec return as output a ranked list of items, and we take the first top-20 topics, and match them with the ground-truth

[20] https://github.com/simov/markdown-viewer

Table 5.7 Recommendation results for the *markdown-viewer* repository (topics matching with ground-truth data are reported in bold).

Original topics (D_2)	**Freq. (D_2)**	**Processed topics (D_3)**	**Freq. (D_3)**	**Rank**	**TopFilter$^+$**	**HybridRec**
javascript	865	javascript	884	1	**markdown**	**markdown**
chrome-extension	69	chrome	123	2	**firefox**	emoji
markdown	105	markdown	123	3	sass	**browser**
firefox	65	firefox	68	4	vim	**chrome**
chrome	103	extension	38	5	github-api	emacs
firefox-addon	29	viewer	34	6	c	html
firefox-extension	17	addon	38	7	editor	**javascript**
browser-extension	13	browser	220	8	**javascript**	**extension**
markdown-viewer	4			9	html	**firefox**
				10	vue	mozilla
				11	document	privacy
				12	react	safari
				13	android	golang
				14	git	opera
				15	nodejs	**addon**
				16	library	python
				17	emoji	editor
				18	web	react
				19	python	secure
				20	book	latex

Table 5.8 Success rate, precision, and recall.

	Success rate		Precision		Recall	
N	TopFilter$^+$	HybridRec	TopFilter$^+$	HybridRec	TopFilter$^+$	HybridRec
1	**0.171**	0.153	0.113	**0.212**	0.014	**0.034**
2	0.223	**0.236**	0.077	**0.179**	0.018	**0.056**
3	0.258	**0.291**	0.060	**0.153**	0.021	**0.070**
4	0.276	**0.317**	0.052	**0.133**	0.024	**0.082**
5	0.290	**0.335**	0.044	**0.117**	0.025	**0.090**
6	0.382	**0.504**	0.084	**0.133**	0.058	**0.122**
7	0.417	**0.549**	0.099	**0.137**	0.079	**0.145**
8	0.449	**0.579**	0.101	**0.136**	0.091	**0.164**
9	0.465	**0.598**	0.099	**0.132**	0.100	**0.179**
10	0.479	**0.614**	0.095	**0.127**	0.108	**0.191**

data. It is evident that HybridRec provides more matched items compared to TopFilter$^+$. In the following experiments we show that the preprocessing steps reduce the usage of different terms that have a very close semantics and increase the topic frequency.

The values of the considered quality metrics, i.e., success rate, precision, recall, top rank, and catalog coverage, reported in the next subsections are obtained by calculating their average values on all the folds from the cross-validation process.

RQ$_1$: *How does HybridRec perform compared to TopFilter$^+$?* We compare HybridRec with TopFilter$^+$ on all the considered datasets, i.e., D_1, D_2, and D_3. Given a testing project p, a certain number of topics τ is used as input, and the remaining ones are saved as ground truth

data *GT(p)*. In our experiments, τ is always considered half of the number of topics already assigned to the project under analysis, and the number of neighbor projects k is set to 20. This was identified as the configuration that brings the best performance to TopFilter$^+$ [66]. Moreover, we try with different values of N to find out how the size of recommended items impacts the prediction performance.

The average success rate, precision, and recall scores obtained by running the ten-fold cross-validation technique with HybridRec and TopFilter$^+$ on D_1, D_2 and D_3 are reported in Table 5.8, considering consecutive cut-off values, i.e., $N = \{1, \ldots, 10\}$. In the table, a better performance – corresponding to higher scores – is printed in bold. Altogether, it is evident that HybridRec outperforms TopFilter$^+$ for any value of N except the success rate for $N = 1$. For instance, the best success rate obtained by TopFilter$^+$ is 0.479 when $N = 10$, while the corresponding score by HybridRec is 0.614. Moreover, the maximum precision score is 0.212 for HybridRec, which is much better than 0.112, the maximum precision achieved with TopFilter$^+$. Concerning recall, though both approaches yield a considerably low performance, HybridRec still always outperforms TopFilter$^+$ by all the cut-off values N. We find out in RQ_4 when HybridRec can improve its prediction performance.

Answer to RQ_1. In comparison to the baseline TopFilter$^+$, HybridRec yields a substantial performance improvement in terms of success rate, precision, and recall.

RQ_2: *In comparison to MNBN, does CNBN contribute to a better HybridRec performance?* In our previous work [66], we utilized a balanced dataset manually curated from GitHub to improve MNBN's performance as the network is not able to handle unbalanced datasets. Nevertheless, this does not essentially resemble a real-world situation in the context of OSS platforms, where the distribution of topics and repositories is usually not balanced. Thus, we employ an enhanced version of MNBN, namely CNBN, to deal with realistic settings. This research question aims to validate such a hypothesis. To compare with MNBN, we run HybridRec using only the **ST** component (see Section 5.2.1), without triggering the subsequent collaborative filtering phase.

Figure 5.13 shows the success rate considering different cut-off values, i.e., $N = \{1, \ldots, 10\}$. It is evident that using CNBN obtains a better success rate with compared to using MNBN, given that an unbalanced dataset is considered. For instance, the improvement is around 10% with $N = 2$, i.e., MNBN and CNBN achieve 0.50 and 0.67 respectively in recommending at least one correct topic. As expected, increasing the number of recommended items leads to a better performance, i.e., CNBN's success rate reaches 0.90 with $N = 9$ and $N = 10$. This

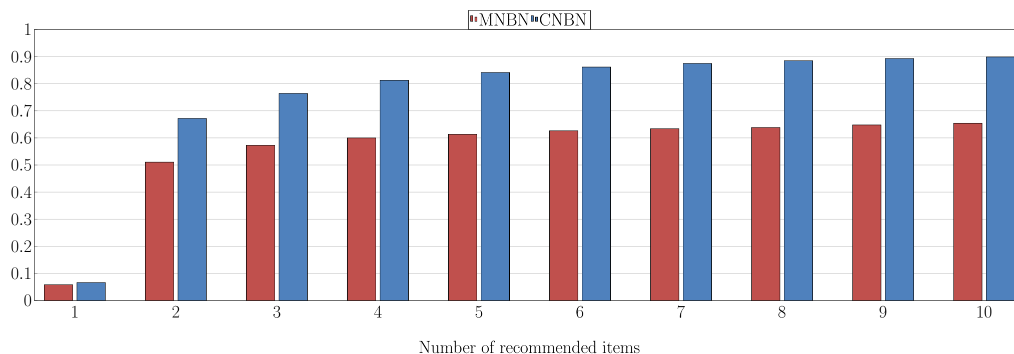


Fig. 5.13 Success rate for D_1 considering $N = \{1, \ldots, 10\}$.

holds also for the MNBN, which however achieves worst performance since its success rate is always lower than 0.65.

Table 5.9 Precision, recall with D_1.

	Precision		**Recall**	
N	MNBN	CNBN	MNBN	CNBN
1	0.582	**0.663**	0.323	**0.374**
2	0.266	**0.350**	0.298	**0.401**
3	0.231	**0.333**	0.362	**0.529**
4	0.194	**0.292**	0.392	**0.597**
5	0.164	**0.257**	0.408	**0.641**
6	0.143	**0.227**	0.422	**0.673**
7	0.126	**0.203**	0.431	**0.694**
8	0.112	**0.184**	0.437	**0.714**
9	0.102	**0.168**	0.447	**0.728**
10	0.941	**0.154**	0.454	**0.741**

To better study the performance of both techniques, i.e., MNBN and CNBN, we compute the precision and recall scores and show them in Table 5.9. Overall, CNBN improves the results obtained by MNBN for all the considered metrics. By considering all the cut-off values, the precision scores are increased by 10% on average. In particular, precision reaches its maximum at N=2, i.e., 26.57 and 35.00 for MNBN and CNBN, respectively; The minimum value obtained by MNBN is 9.41, while the corresponding score by CNBN is 15.44. Concerning recall, we observe a remarkable improvement when CNBN is adopted, i.e., it increases MNBN's results 30% of the time with $N = 10$. As expected, increasing the number of the returned items positively affects the performance of both networks. Still, CNBN improves recall from 40.14 to 74.07, while MNBN gains 45.38 as its maximum value. In other words, the usage of CNBN reduces the impact of false negatives during the prediction phase. Finally, the advantage of CNBN is eventually confirmed by the Top-rank metric, which measures the accuracy of the two networks in recommending the first item,

i.e., the most probable one. In particular, the computation shows that CNBN obtains a better performance also in this case by increasing the top rank prediction up to 65.85 while MNBN gets only 49.55.

> **Answer to RQ$_2$.** On an unbalanced dataset, compared to MNBN, CNBN improves the prediction performance. As data sources are unbalanced by their nature, adopting CNBN helps obtain a more precise prediction in the field.

RQ$_3$: *How do the preprocessing steps impact on the HybridRec performance?* We measure the impact of the preprocessing steps on the collaborative filtering recommendation engine on the three datasets collected from GitHub, i.e., D$_1$, D$_2$ and D$_3$. As described in Section 5.2.2, D$_2$ is obtained from D$_1$ by filtering out topics that occur in less than 20 repositories, while D$_3$ is extracted from D$_1$ by applying the preprocessing rules defined in Section 5.2.1. To compare with the original TopFilter approach [66], we run HybridRec using only the **CF** component (see Section 5.2.1).

Given a testing project p, a certain number of topics is used as input, i.e., $\tau = 5$, and the remaining ones are saved as ground truth data, i.e., *GT(p)* (cf. Section 5.2.2). Moreover, we investigate various number of recommended items $N = \{5, 10, 15, 20\}$ to understand how the size of recommended items impacts the prediction performance of TopFilter. The average success rates obtained by running the ten-fold cross-validation technique with HybridRec on D$_2$ and D$_3$ are depicted in Fig. 5.14.

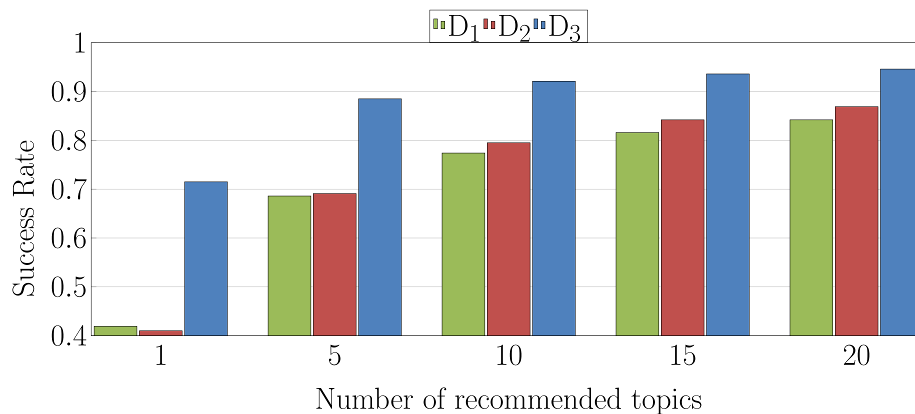


Fig. 5.14 Success rate values for $N = \{1, 5, 10, 15, 20\}$.

It is evident that HybridRec yields a better success rate when preprocessed datasets, i.e., D$_2$ and D$_3$, are considered. For instance, it gets SR@1 of 0.715 for D$_3$, while for other datasets, the corresponding value is always lower than 0.42. Moreover, by comparing the result given by considering different cut-off values N, we realize that there is no significant

difference between the results for $N = 10$ to $N = 20$. This means that most of the matched items concentrate on the top-5 ranked list, and considering a longer list of items does not bring any positive matches.

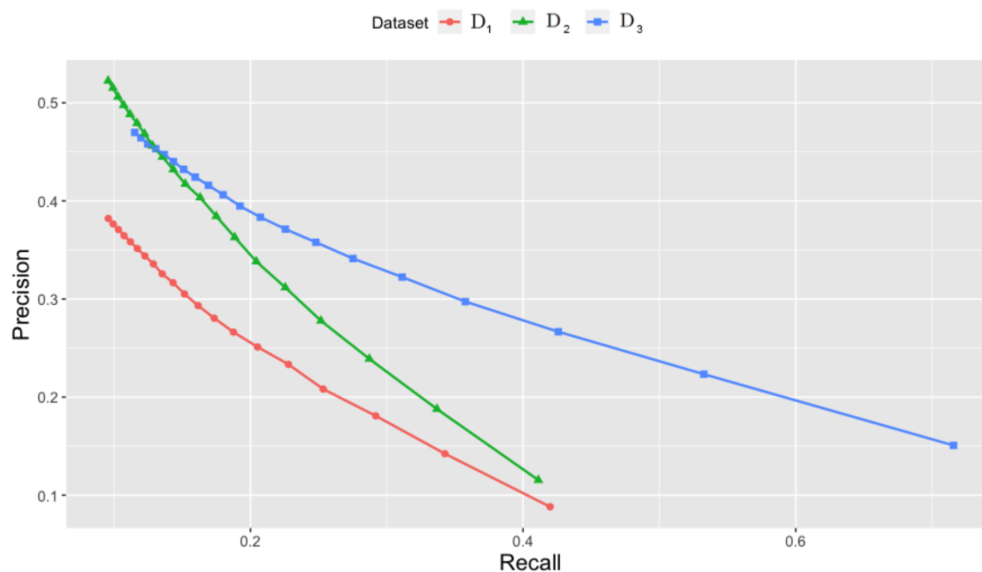


Fig. 5.15 Precision/recall curves.

To further study the performance of HybridRec, we depict in Fig. 5.15 the precision/recall curves (PRCs). For this setting, we varied the recommended items N from 1 to 20, aiming to study the performance for a more detailed recommendation list. Each dot in a curve represents the precision and recall scores obtained for a specific value of N. Furthermore, we fixed $k = 20$ since this number of neighbors brings the best prediction outcomes among others, while it allows HybridRec to maintain a reasonable execution time. As a PRC close to the upper right corner corresponds to a higher precision and recall, suggesting a better performance [182], Fig. 5.15 shows that by considering a preprocessed dataset, HybridRec gains a better prediction. In particular, the worst precision-recall relationship is seen by the raw dataset D_1, while DGP obtains the best one. Overall, these results are consistent with those presented in Fig. 5.14, i.e., using the preprocessing steps given in Section 5.2.1 enables HybridRec to enhance its performance substantially.

Finally, we investigate if HybridRec can provide a wide range of topics to repositories, taking into consideration catalog coverage. This metric measures the percentage of the recommended topics in the training data that the model recommends to a test set, and a higher value corresponds to better coverage. Table 5.10 reports the average coverage values got from running HybridRec on the datasets, i.e., D_1, D_2, and D_3. The table suggests an evident outcome: we get better coverage by considering longer lists of items. We also studied the

Table 5.10 Catalog coverage.

N	D_1	D_2	D_3
1	0.013	0.202	0.006
2	0.022	0.321	0.017
3	0.030	0.411	0.026
4	0.038	0.483	0.033
5	0.045	0.551	0.041
6	0.053	0.604	0.049
7	0.060	0.647	0.056
8	0.068	0.694	0.063
9	0.076	0.732	0.070
10	0.084	0.765	0.077

impact of the preprocessing step on the coverage results. The catalog values range from 0.013 (obtained for D_1 with $N = 1$) to 0.765 (obtained for D_2 with $N = 20$). It is important to recall that D_2 and D_3 are very different with respect to the number of topics, i.e., 455 compared to $6,442$. At the same time, the sets of considered repositories are very similar (only 239 projects are discarded in D_2) (see Section 5.2.1). Altogether, we see that using a denser dataset for training, i.e., projects with more topics is beneficial to success rate, accuracy but not to catalog coverage.

Answer to RQ$_3$. The preprocessing steps are beneficial to the recommendation process as they allow HybridRec to retrieve more relevant topics, thus improving the recommendation performance.

RQ$_4$: *What are the key differences between GitHub and MVN Repository, and how do they impact on the whole HybridRec recommendation process?* To investigate HybridRec's performance in a different type of repositories, we run it on a different dataset, namely D_M collected from MVN Repository (see Section 5.2.2). The artifacts available in MVN Repository are labeled with *tags*, which express similar concepts to GitHub topics, i.e., summarizing their key functionalities. Moreover, each artifact comes with a textual description that can be used as a README file.

There are differences in the nature of topics in GitHub with respect to tags in MVN Repository. In particular, topics for a GitHub repositories are manually assigned by the owner(s). Thus, the resulting set of the given topics might include issues, e.g., duplicate drop, and word spelling, which necessarily require some pre-processing steps. In our proposed approach, we refined the mined GitHub topics by means of the rewriting rules defined in Section 5.2.1. In contrast, MVN Repository tags are well-maintained as they are not freely assigned by developers but presumably audited by a watchdog. This implies that MVN

repositories are more curated than those hosted in GitHub, thus limiting errors that might happen due to the manual activities performed by developers. Altogether, this helps enhance the list of possible tags by automatically suggesting new tags, thus improving the reachability of the artifacts.

We conduct experiments for the D_M dataset using the same settings applied on the GitHub datasets and compute the quality metrics accordingly. Figure 5.16 represents the success rate scores obtained by running HybridRec on the Maven dataset, using different values of k, the number of neighbour projects (see Section 5.2.1). As we can see, the usage of a more curated dataset to make predictions contributes to improving the overall performance, i.e., the maximum success rate obtained by HybridRec reaches almost 0.90 by most of the cut-off values, and eventually, it goes beyond the 0.90 threshold with $N = 10$.

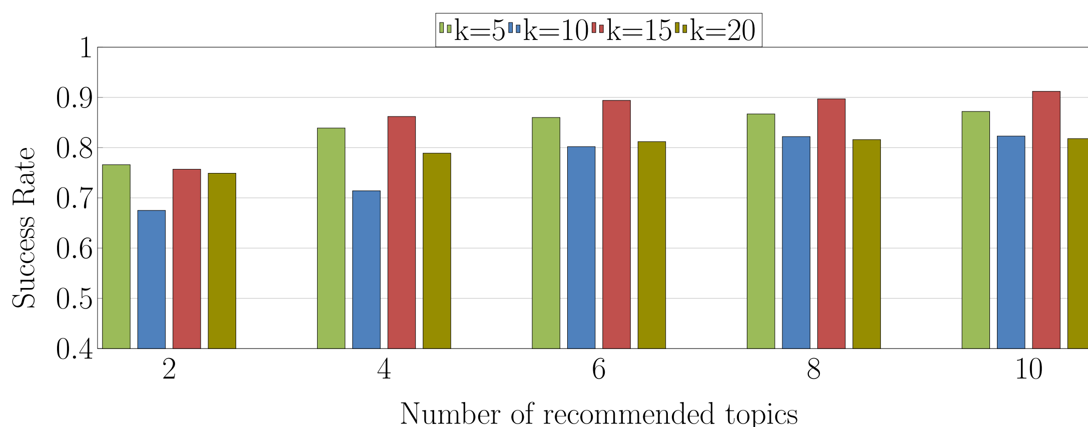


Fig. 5.16 Success rate on MVN Repository dataset.

Concerning the neighborhood parameter, the results show that the best success rate is achieved with $k = 15$. This means that starting from $k = 15$, adding more neighbour projects to compute recommendations does not bring any matched tags. Overall, running HybridRec on D_M yields a better prediction performance than the results obtained with the D_1 dataset. It is worth noting that no preprocessing has been applied on Maven tags since the hand-written rules are tailored for GitHub topics that need some cleaning due to duplicates or incorrect terms. In contrast, D_M does not require such a process since tags are already fine-tuned, and their constituent terms are concretely defined. For instance, there is no need to predict the language programming tag since all projects are written in Java.

We compute precision and recall scores and show them in Fig. 5.17. Similar to the previous results, running HybridRec on D_M contributes to a better performance since the precision and recall are higher, compared to those obtained with the D_1 dataset in Table 5.8. The precision scores increase by 10% on average with the best configuration, i.e., $k = 5$.

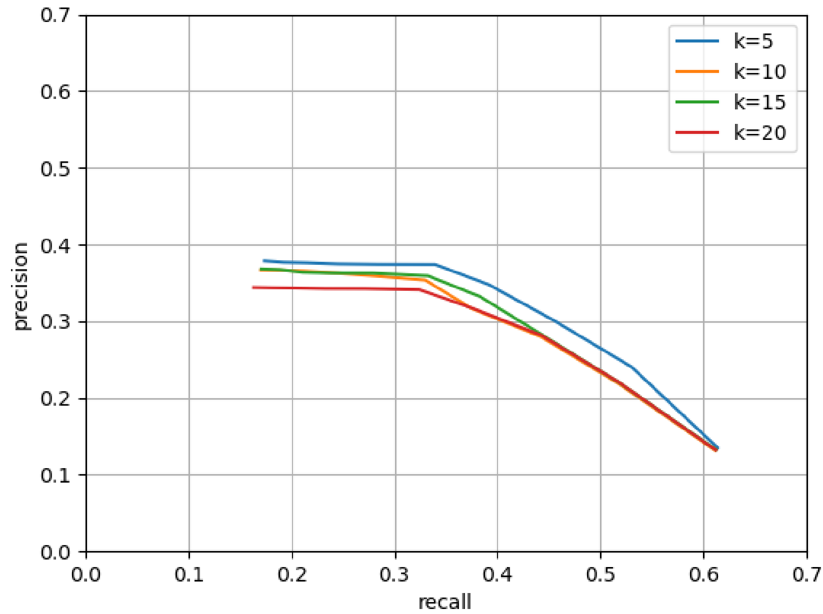


Fig. 5.17 Precision recall curve on the MVN Repository dataset.

Similarly, the recall scores improve from 0.191 to 0.70 when more recommended items are considered in the ranked list.

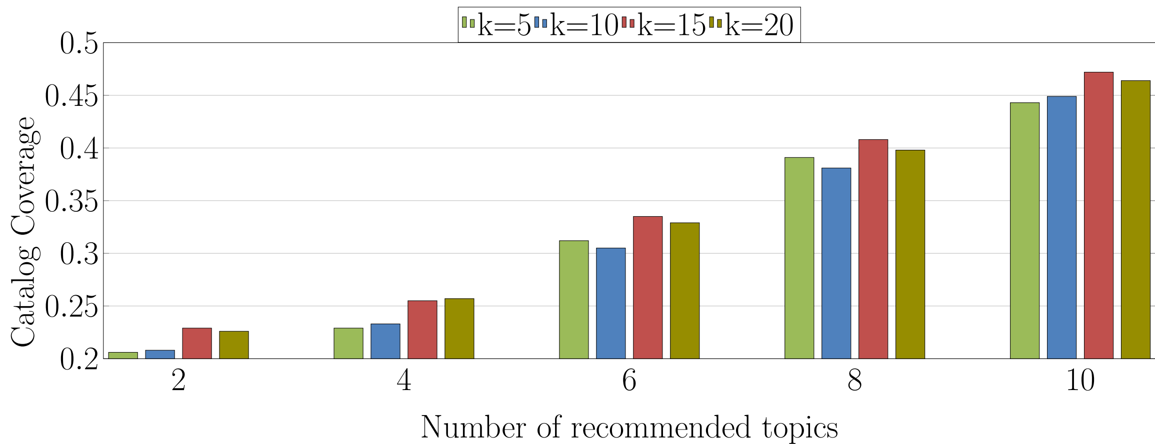


Fig. 5.18 Catalog coverage on the MVN Repository dataset.

The gain in performance is further confirmed with the catalog coverage scores in Fig. 5.18. HybridRec achieves a maximum catalog coverage of 0.47 with $N = 10$. This means that the probability that HybridRec recommends correct items is higher if D_M is considered. Referring to Table 5.6, we see that the number of topics in D_M is much smaller than that of

D_1, i.e., 489 compared to 19,337. Similarly, D_M contains less projects than D_1, i.e., 2,932 compared to 11,694. Altogether, this demonstrates that even on a small Maven dataset, HybridRec can still obtain a good performance.

Answer to RQ$_4$. Compared to topics in GitHub, tags in MVN Repository are more well-defined as they are audited by a central authority. Due to this reason, running HybridRec on input data curated from Maven repositories brings in a better prediction performance.

5.2.4.2 Discussion

HybridRec has been developed by combining a complement naïve bayesian network with a collaborative-filtering technique. Moreover, we employed a tailored preprocessing phase on the considered topics to clean and refine the input data before feeding it to the recommendation engine. Altogether, the boosting mechanisms help HybridRec retrieve more relevant topics. The empirical evaluation has shown that HybridRec is able to improve the recommendation results compared to the baselines.

In this section, by relying on the previously performed experiments, we analyze the five challenges highlighted in Section 5.1.1.

Concerning Challenge **C1** (*data redundancy*), we have the following investigations. Topics for a GitHub project are manually assigned by the owner(s). Thus, the resulting set of given topics might include issues (e.g., duplicate drop, and word spelling) that necessarily require some pre-processing steps. In the proposed approaches, we refined the mined GitHub topics by means of NLP techniques and word embeddings, e.g., GloVe [200], word2vec [162], and FastText [98]. Furthermore, we adopt a well-structured set of rules to consider frequent patterns as well as to refine them. We have proven that such pre-processing steps improve the overall HybridRec's prediction capabilities.

Another aspect that may impact the outcomes is the topic's *distribution*. To analyze to what extent this can affect the recommendation items, we make use of a GitHub dataset employed in our previous work [66]. Fig. 5.19 gives an overview on the distribution of topics among repositories for the raw D_1 dataset. From this picture, we can observe that many topics are infrequent, i.e., 14,175 among 15,743 topics are used in less than 11 projects, and just a few topics are widely adopted by more than 200 projects. In other words, the long tail effect has a profound impact on the GitHub datasets.

Challenge **C2** involves the *structure of available metadata* that an OSS ecosystem can provide to perform the recommendation activities. A GitHub repository offers a lot of metadata about the activities of developers as well as their interactions. For instance, the platform keeps track of any user who makes a pull request or modifies the project by adding

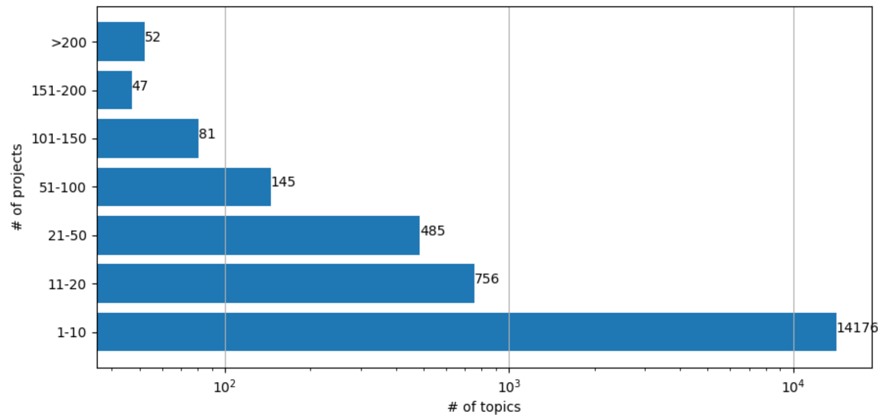


Fig. 5.19 The distribution of topics in the D_1 dataset.

files through commits.[21] Furthermore, each repository is characterized by textual information, e.g., README files, or Wiki content. In other words, OSS mining tasks can be characterized by two dimensions: the *employed recommendation technique* and the proper *metadata* that can be extracted from the input data.

Table 5.11 Metadata used by the **ST** and **CF** components.

	GitHub		
Component	Topics	Featured Topics	README files
ST	✓	✓	✓
CF	✓	✗	✗

Table 5.11 describes these two dimensions in terms of the techniques and datasets presented in this work. As shown in the table, the collaborative-filtering component relies solely on the list of initial terms to perform the recommendations considering GitHub platform since all needed relations concerning projects are encoded in the matrix used to fed the recommendation engine. Furthermore, it employs the initial list of terms to enable the recommendation engine. As stated in Section 5.2.2, using a larger number of inputs definitively improves the tool's performance. In contrast, the **ST** component needs textual files to predict GitHub topics, namely README files.

In the context of collaborative development, the availability of *popularity mechanisms* play an important role to foster the project's visibility. As we mentioned in the description of **C3**, GitHub is reliant on a well-defined popularity mechanism that considers stars, forks, and featured topics. Both the **ST** and **CF** components benefit from using popular projects as the initial dataset, i.e., we observed the best results with respect to various quality metrics.

[21] https://docs.github.com/en/free-pro-team@latest/github/collaborating-with-issues-and-pull-requests/about-pull-requests

Another important factor to consider is the fact the GitHub exposes the selected featured topics in a dedicated repository[22] and web pages.[23] Despite this, the crawling phase requires a well-structured process to extract the needed information properly.

Concerning **C4** (*Crawling and Data Dump*), we see that even when the required data is available, the gathering process could be a daunting task. *Data dumps* can simplify this activity by offering all the information in a structured way, for instance with a database. As we mentioned before, data dumps for GitHub have been provided by an external tool, i.e., GHTorrent [97]. Nevertheless, data contained in these collections is not suitable for supporting automatic tagging activities. Therefore, we made use of a crawler that can download the data belonging to the repositories. In our previous studies, we exploited the PyGitHub library since GitHub offers all the required APIs.

Table 5.12 Configuration parameters of the **ST** and **CF** components.

Parameters	**ST**	**CF**
Number of considered labels	134	6,422
Number of projects	6,700	5,859
TF-IDF vectorizer	✓	-
Topic preprocessing	✓	✓
GuessLang module	✓	-
Cutoff	-	20
Neighborhood	-	25

With respect to Challenge **C5** (*Configurations of the underpinning recommendation systems*), besides the chosen OSS platform, the capabilities of the underpinning recommendation system are affected by the internal parameter configurations. The possible tuning parameters depend on the algorithm employed to perform the recommendation as well as the metadata offered by the platform. Table 5.12 describes the parameters employed by the two considered components, i.e., ST and CF, to bring the best prediction performance.

As described in Section 5.2.4.1, we have these values to reach a balanced dataset which is crucial to earn a better performance. This process has been conducted manually for the GitHub datasets as they exhibit heterogeneous support for each featured topic in terms of projects. Concerning the collaborative-filtering component, this consideration mainly impacts the selection of the cut-off values t. In particular, to cope with different impacts of the long-tail effect, we varied t, for example, to evaluate the **CF** component with the GitHub dataset we changed t from 5 to 20 with 5 as the step, while t has been shifted from 1 to 5 with 1 as the step. Moreover, because of the different levels of similarities between the projects

[22] https://github.com/github/explore
[23] https://github.com/topics

of GitHub we used a different number of neighbors k when TopFilter recommends suitable labels.

5.2.5 Threats to validity

In this section, we highlight possible threats that may have an impact on the outcomes of the conducted experiments. We also list the countermeasures adopted to mitigate these issues.

Internal validity The overall recommendation capabilities of the presented approaches could be compromised by the dataset features, i.e., the number of unique tags, lack of support for each category. To cope with these risks, we analyzed several configurations obtained by means of a well-defined data preprocessing phase. These settings allow for a more comprehensive evaluation of the two approaches.

External validity As specified in Section 5.2.2, the data extraction phase has been conducted by relying on the GitHub API. This could compromise the quality of the obtained information as we cannot access the entire knowledge due to the rate limits. Such a threat has been mitigated by considering popular repositories. In this way, we chose the most representative projects that help minimize any potential bias.

Construction validity The comparison of TopFilter$^+$ with HybridRec might be susceptible to bias. We carefully examined existing work in the domain, but no comparable tool is available with an available sound replication package. Thus, we used the same evaluation conducted to evaluate MNBN in the original work, bringing the best results. We adapted the structure of the two techniques to mitigate any possible pitfalls in the overall evaluation process.

5.3 Conclusion

OSS platforms play an important role in aggregating and handling projects developed by a prolific community. Automatic tagging is one of the most valuable techniques to improve their discoverability, even though it requires a lot of effort to produce useful outcomes.

In this chapter, we have presented first an approach to recommend a set of featured topics given a software project endowed with a corresponding README file. The tool is based on a probabilistic machine learning network, the Naïve Bayesian classifier. We encode the relevant information about repositories using the TF-IDF weight scheme. After the training phase, the approach provides the user with a list of featured topics related to its project. We evaluated the approach using cross-fold validation.

To address the MNBN limitations, we conceived HybridRec, a hybrid recommender system working on top of an enhanced version of the stochastic network and a collaborative-filtering technique to recommend topics. We performed an empirical evaluation on real-world datasets to study HybridRec by comparing it with state-of-the-art toots. The results showed that the newly conceived approach improves our former recommender systems substantially. More importantly, we demonstrated that HybridRec can increase its prediction performance on well-curated data sources.

Chapter 6

Assisting modelers in specifying models and metamodels

The deployment of model-driven engineering (MDE) techniques necessitates advanced tools to facilitate various modeling activities [176, 185]. Among others, there is the need to specify metamodels, models and the development of model analysis and management operations. Nevertheless, existing tools such as those that are based on Eclipse EMF[1] normally offer only canonical functionalities, i.e., drag-and-drop, specification of graphical components, auto-completion, and they do not support context-related recommendations, which may come in handy for modelers to complete their tasks.

In this respect, intelligent modeling assistants (IMAs) [171] have been recently proposed to support modelers during their daily activities. Most of the existing tools employ technologies such as neural networks and NLP techniques to automatize the whole design process [42, 284]. Altogether, this aims to facilitate the completion of metamodels by providing modelers with insightful artifacts, such as attributes or relationships. Nonetheless, there are still open challenges to be tackled, e.g., offering a convenient way to specify metamodels and models, covering different application domains.

This chapter presents two different approaches conceived to support model and metamodel specification that relies on two completely different techniques. First, we present MemoRec [69], a recommender system that exploits a context-aware collaborative filtering technique [49] to recommend relevant artifacts related to the modeling domain. MemoRec exploits four different encoding techniques (i.e., different selections of what information has to be kept from a metamodel) to preprocess the input data. More importantly, we tailor the internal design to compute similarity among metamodels in an efficient way. Given a

[1] https://www.eclipse.org/modeling/emf/

metamodel partially specified by the modeler, MemoRec is able to suggest two types of artifacts, namely *(i)* metaclasses at the level of package; and *(ii)* structural features for a given metaclass. This work has been published in the Software and System journal [69] and the candidate contributes mostly in conceiving the encoding schemes.

Afterward, we present MORGAN, a MOdeling Recommender system based on GrAph Kernels to support the completion of both models and metamodels. As for MemoRec, this work has been published in the Software and Systems Modeling journal [76]. The candidate curates the design, implementation, and evaluation of MORGAN. As the first step, MORGAN produces a textual representation of the considered (meta)model by employing tailored model parsers. Then, relevant features are encoded in a graph-based representation that preserves the internal structure of the represented model, i.e., relationships among defined entities. Such an encoding process is model-agnostic since we produce textual files from different formats commonly used to represent (meta)models, i.e., *ecore*, *xmi*. Afterward, the underpinning GNN model computes a graph kernel function used to assess the similarity among nodes by relying on the extracted graphs.

Furthermore, MORGAN supports the completion of models expressed in Javascript Object Notation[2] (JSON). Such a format is used to represent data in a structured way, fostering the interchangeability of valuable information. Even with the introduction of JSON schema meta-language [5], it is still essential to assist developers during their specification. In this respect, a recent work presents a bridge between the JSON schema and MDE metamodels, suggesting that MDE techniques can come in handy in designing such models [53]. Thus, we introduce the support for JSON schema completion by relying on a tailored parser to encode these specifications in a graph-based structure. Furthermore, we overhaul the underlying recommendation engine by introducing *(i)* a lemmatization preprocessing step to improve MORGAN's encoding; and *(ii)* an enhanced graph kernel function by relying on a modified version of the Vertex Histogram technique [251] that exploits a frequency matrix. To enable the recommendation on the new dataset, we mined JSON schema from GitHub repositories by using a well-defined crawler. Afterward, the relevant features are represented in a graph-based structure to encode information represented by the model, e.g., relationships among defined entities, the name of the attributes, to list a few. Such an encoding process is model-agnostic since we produce textual files from different formats commonly used to represent. MORGAN eventually suggests relevant artifacts including the JSON schema elements, demonstrating the generalizability of the system.

[2] https://www.w3.org/TR/json-ld11/

Outline of the chapter: The first part of the chapter introduce the problem of model assistance by describing three different motivating examples in Section 6.1.1. Section 6.1.2 overviews the MemoRec architecture and the tool is evaluated in Section 6.1.3. We report the obtained results in Section 6.1.4 while possible threats to validity are discussed in Section 6.1.5. The second part of the chapter is dedicated to MORGAN. Section 6.2.1 discusses the underpinning techniques employed by MORGAN, i.e., the graph kernel similarity. The approach is presented in Section 6.2.2. Section 6.2.3 and Section 6.2.4 report the evaluation settings and the obtained results respectively. We eventually highlight the limitations of the tool and possible mitigation actions in Section 6.2.5. The conclusion of the chapter are presented in Section 6.3.

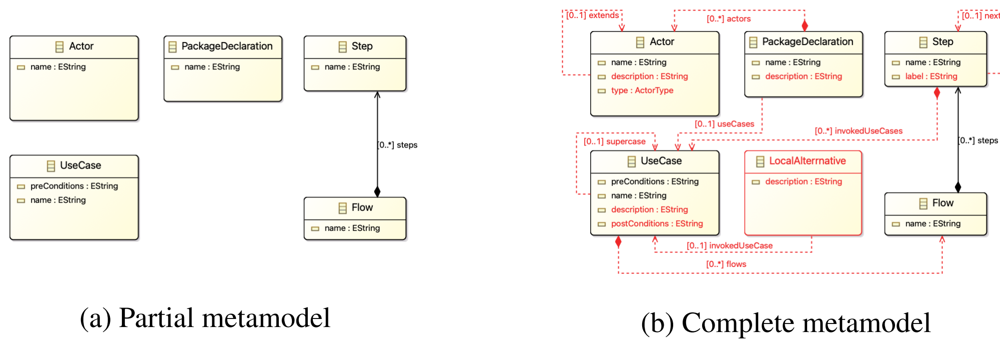
6.1 MemoRec

6.1.1 Motivation and background

We simulate a modeler who is specifying a metamodel/model, and at certain point in time, due to either the complexity of the required task or the lack of experience, he/she does not know how to proceed. In this respect, a model assistant is expected to help enhance the specification of a partial model by recommending new elements as illustrated below.

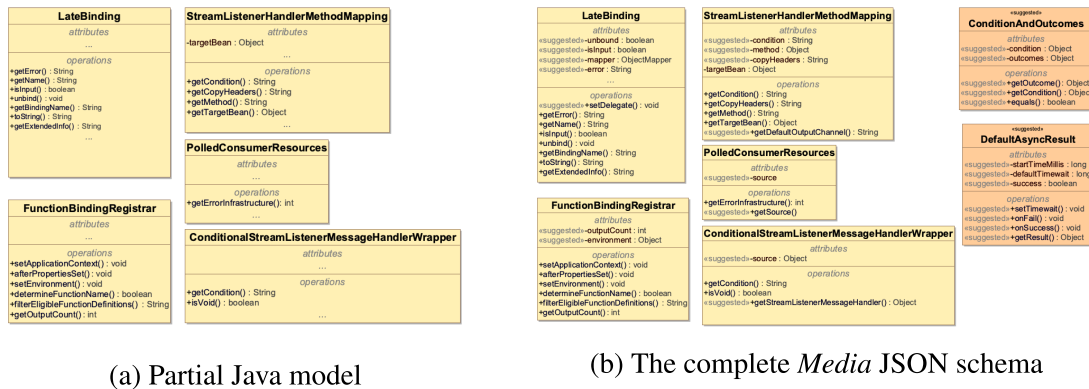
▷ **Metamodeling assistant.** Figure 6.1 represents the explanatory UMLDSL metamodel. At the time of consideration, the modeler has defined only basic attributes and one relation between the *Step* and *Flow* entities. In particular, the initial metamodel specified in Fig. 6.1a defines the key concepts to represent a UML *UseCase*, i.e., *Actor*, *Step*, and *Flow*. Figure 6.1b represents the final version of the corresponding metamodel, where the *PackageDeclaration* entity is completed with *use cases* and *actors* references, and each use case can have several *flows* composed of a sequence of *steps*. By comparing the two sub-figures, we see that some assistances are needed to enrich the partial metamodel shown in Fig.6.1a, e.g., by adding new classes, attributes or relations. Moreover, it is also necessary to suggest new metaclasses including structural features, i.e., attributes and references. For instance, as seen in Fig. 6.1b, the final *LocalAlternative* metaclass includes the *description* and *includedUseCase* structural features, and this makes the original metaclass more informative/complete. In this way, the assistant should be able to provide the modeler with useful recommendations to help in finalizing the metamodeling activities.

In fact, metamodels are used to represent concepts at a high level of abstraction. To design real systems, MDE makes use of models that usually conform to a corresponding



(a) Partial metamodel (b) Complete metamodel

Fig. 6.1 The illustrative UMLDSL metamodel.



(a) Partial Java model (b) The complete *Media* JSON schema

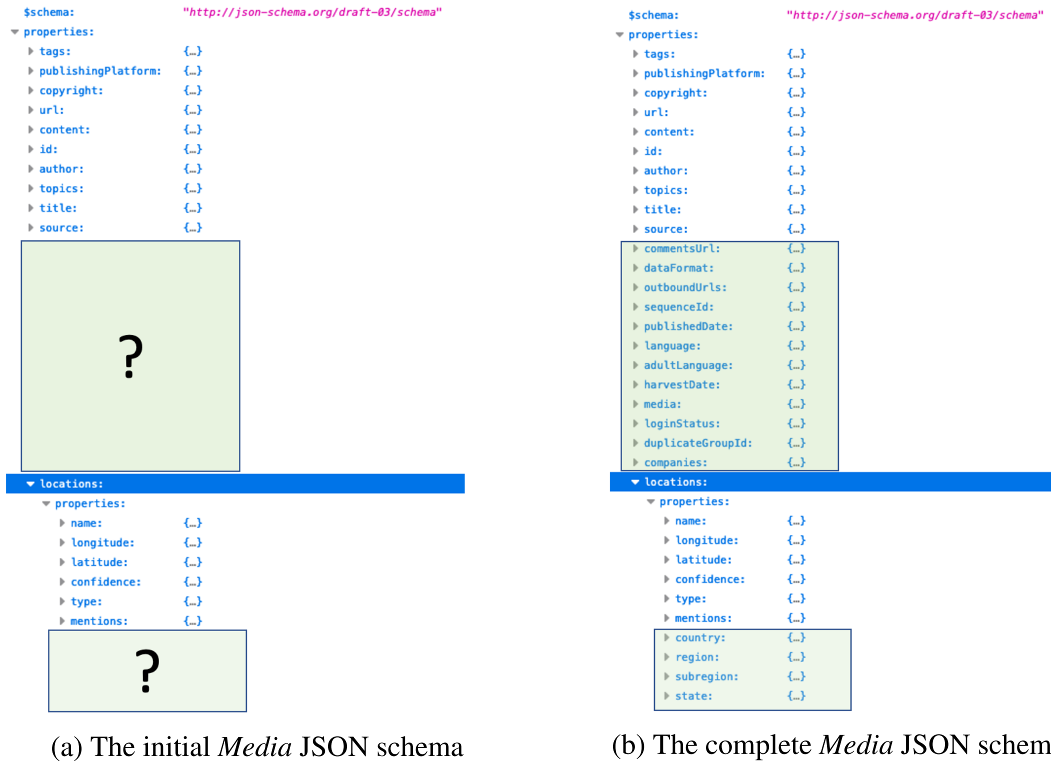
Fig. 6.2 The explanatory SPRINGBOOT model.

metamodel. As a result, recommending additional elements to be incorporated in modeling activities is also meaningful.

▷ **Modeling assistant.** We make use of the MoDisco framework[3] to extract models from compilable Eclipse projects. As an example, we show in Fig. 6.2a an excerpt of the Java model of the Spring bootproject[4] parsed by MoDisco. The model conforms to the MoDisco Java metamodel, covering the full Java language and constructs, i.e., packages, classes, methods, and fields. In this way, existing Java programs can be represented as MoDisco Java models. In particular, Fig. 6.2a depicts a partial Java model with the following five classes:

1. *LateBinding*;
2. *FunctionalBindingRegistrar*;
3. *StreamListenerHandlerMethodMapping*;

[3] https://www.eclipse.org/MoDisco/
[4] https://spring.io/projects/spring-boot



(a) The initial *Media* JSON schema (b) The complete *Media* JSON schema

Fig. 6.3 The explanatory APACHE STREEMS PROVIDER JSON Schema.

4. *PolledComsumerResources*; and
5. *ConditionalStramListenerMessageHandlerWrapper*.

All these classes are incomplete at the time of consideration, and the modeler is supposed to add the missing items. For the sake of presentation, we show the general structure of the Spring boot Java model in terms of classes, methods, and fields. As it can be seen, similar to the above mentioned UMLDSL metamodel, an incomplete Java model can be enriched with new classes, methods, and fields. In Fig. 6.2b, we represent a set of possible model elements to complete the partial model. In particular, the model elements tagged by *suggested* are the ones that should be recommended for completing the model. For instance, two classes including their methods and fields are recommended, i.e., *ConditionAndOutcomes* and *DefaultAsyncResult*. Moreover, the model assistant is expected to recommend missing class elements, e.g., the *error* field and the *getError* method for the *LateBinding* class.
▷ **JSON schema assistant.** Similarly, modeling assistants can be employed to complete a partial JSON schema with relevant properties. Although they are artifacts of different nature, Colantoni *et al.* [53] investigate the potential benefits of mapping JSONware technical space to Modelware one, with the aim of making use of languages and documents exchangeable.

Their approach supports language engineers, domain experts, and tool providers in editing, validating, and generating tool support with enhanced capabilities for JSON schemas and their documents. The results show that it is possible to derive editing and validation support based on model-driven technologies for JSON Schema-based languages. Therefore, modeling assistants could be employed to support the completion of such kinds of artifacts. Fig. 6.3 shows an explanatory example with the *Apache stream provider*[5] JSON schema to describe a *Media* entity. Given the partial schema depicted in Fig. 6.3a, we mimic the situation where the modeler has specified a set of initial 11 properties, e.g., tags, copyright, url, author to name a few. At this point, an automated system might be used to fill the partial JSON schema with additional items, such as new properties at schema level or nested ones. In Fig. 6.3b, we devise a possible set of recommendations that includes additional properties, e.g., *dataFormat*, *language*, *adultLanguage*. Furthermore, the *locations* structural property can be enriched with concepts that can increase the expressiveness of the whole schema, i.e., country, region, subregion, and state.

The presented use cases raise the need for a model assistant to support the completion of partially defined metamodels, models, and JSON schema. Since modeling activities are strongly bounded by the domain, a modeler who has limited knowledge of it may encounter some difficulties.

In particular, MemoRec has been specifically conceived to support the completion of metamodels while MORGAN is capable of covering the specification of all types of cosidered modeling artifacts, including models and JSON schema.

6.1.2 MemoRec architecture

6.1.2.1 Context-aware collaborative filtering technique

A context-aware collaborative filtering recommender system provides recommendations to users items that have been bought by similar users in similar contexts [231, 235]. Based on this premise, we successfully developed a recommender system named FOCUS to provide developers with API function calls and usage patterns [181]. We modeled the mutual relationships among projects using a tensor and mined API usage from the most similar projects.

The successful deployment of different variants of collaborative filtering techniques allows us to transfer the acquired knowledge into the MDE domain. We build MemoRec by redesigning and customizing FOCUS to support the completion of metamodels. Mining from similar objects is a building block of collaborative filtering techniques, and we exploit this to

[5]https://streams.apache.org/

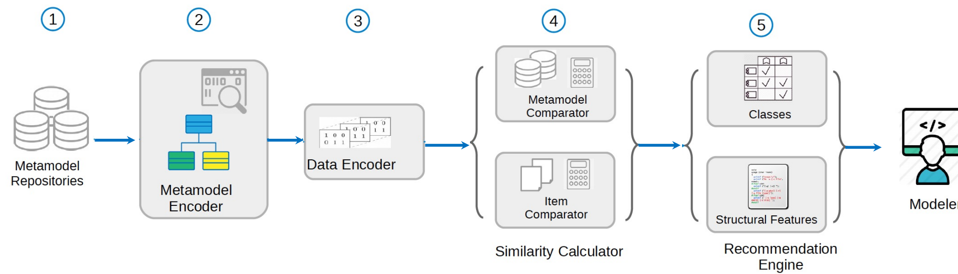


Fig. 6.4 Overview of MemoRec's architecture.

provide recommendations for metamodels under development. In particular, our approach works based on the assumption that: *"if metamodels share some common artifacts, then they should probably have other common artifacts."* In this respect, MemoRec mines meta-classes and structural features from similar metamodels, given an input metamodel. The following section presents our conceived approach to recommend useful elements for a metamodel being under development.

MemoRec provides modelers with recommendations, can be helpful while defining a metamodel, by considering the existing portion of the metamodel as active context. The system makes use of a graph representation to encode the relationships among various metamodels artifacts, and generates recommendations employing a context-aware collaborative filtering technique [235]. In addition, we exploit a tailored textual representation of metamodels, which are encoded to enable the extraction of the containing knowledge to feed as input for the recommendation engine.

The architecture of MemoRec is depicted in Fig. 6.4. To provide recommendations, MemoRec accepts as input a set of *Metamodel Repositories* ①. Afterwards, the *Metamodel Encoder* component ② extracts packages and classes pairs as well as class and structural feature pairs from the metamodel being developed. *Metamodel Comparator*, a subcomponent of *Similarity Calculator* ③, measures the similarity between the metamodels stored in the repositories and the metamodel under specification. Using the set of metamodels and the information extracted by *Metamodel Encoder*, the *Data Encoder* component ④ computes rating matrices. Given an active context of a metamodel, i.e., package or class, *Item Comparator* computes the similarities between packages and classes. From the similarity scores, *Recommendation Engine* ⑤ generates recommendations, either as a ranked list of classes if the active context is a packages, or a ranked list of structural features if the active context is a class. In the remainder of this section, we present in greater details each of these components.

6.1.2.2 Metamodel Encoder

A metamodel defines the abstract concepts of a domain where concepts, as well as the relationships among them, are expressed by the used modeling infrastructure. In particular, a metamodel consists of *Packages* that aggregate similar concepts expressed by *Classes*. *Classes* consist of structural features, i.e., attributes and references. Moreover, *Classes* can inherit structural features from other classes.

Being inspired by our recent work [176], we employ four encoding schemes to represent different views concerning the terms extracted from packages, classes, and structural features named instance. Each scheme has been used to elicit relevant information from the input metamodels according to different granularity levels. In particular, we make use of two *standard encoding* (SE) scheme for recommending structural features within a context class and two other *improved encoding* (IE) scheme for supporting classes within a package context. In particular, we consider the following definitions:

- SE_s: it includes pairs in the form of *<class_name>#<structural_feature_name>* for each structural feature contained within a class. This encoding scheme is used to suggest additional structural features within a given class context;

- IE_s: it consists of pairs *<class_name>#<structural_feature_name>* for each structural feature contained within a class. Moreover, it includes structural features inherited from the super classes. In our previous work [65], we studied how structural features are used with hierarchies. On one hand, we found out that increasing the number of metaclasses with super-types decreases the average number of structural features directly specified in a metaclass, since structural features are spread through the class hierarchies. On the other hand, the average number of structural features including the inherited ones is uncorrelated with the number of metaclasses with super-types. We anticipate that, by including the inherited structural features in IE_s, we will be able to increase the informative part of the encoding. We use this scheme to suggest additional structural features within a given class context;

- SE_c: it includes pairs *<package_name>#<class_name>* for each class contained within a package. This encoding scheme is utilized in providing additional classes within a given package context;

- IE_c: it flattens packages and classes and encodes classes within a default artificial package. We use this encoding scheme to suggest additional classes within a given package context. Since metamodels consist of few *ePackage*s [65], we envision that a flatten representation of classes, i.e., by bypassing the Package/Class containment can

help MemoRec to consider more metamodels and to extract classes from different top similar metamodels.

An encoding scheme depends on two main factors:

– the purpose of the recommendation; depending on the type of the recommended items (e.g., structural features, classes, specialization/generalization of a metaclass, ifnextchar.etcetc.) the encoding scheme should be tailored to support the identified recommendation goal;

– the prediction performance; the encoding scheme strongly impacts on the prediction performance. For this reason, the identification of suitable encoding scheme is an iterative process where the encodings are incrementally improved to maximize the prediction performance for a specific purpose.

As future work, further encoding schemes could be provided to target different kinds of recommendations. For instance, an encoding scheme representing the inheritance relations between classes could suggest a possible set of generalizations or specializations for a given metaclass in the active context. In addition, a different encoding scheme could be used to include types for the recommended structural features.

```
1 Page#title
2 Page#meta
3 Static#Content
4 Static#picture
5 Dynamic#list
6 Dynamic#entity
7 Entity#name
8 Entity#fields
9 Field#isPK
10 Field#name
```

(a) SE_s encoding

```
1 Page#title
2 Page#meta
3 Static#Content
4 Static#picture
5 Static#title
6 Static#meta
7 Dynamic#list
8 Dynamic#entity
9 Dynamic#title
10 Dynamic#meta
11 Entity#name
12 Entity#fields
13 Field#isPK
14 Field#name
```

(b) IE_s encoding

```
1 Web#Page
2 Web#Static
3 Web#Dynamic
4 Data#Entity
5 Data#Field
```

(c) SE_c encoding

```
1 Package#Page
2 Package#Static
3 Package#Dynamic
4 Package#Entity
5 Package#Field
```

(d) IE_c encoding

Fig. 6.5 The Web metamodel data extraction.

Fig. 6.5 depicts an extract of the four encoding schemes. In the following subsection, we show how the pairs *package/class* and *class/structural feature* relationships are encoded. In Section 6.1.3, we evaluated MemoRec by considering the four encoding schemes described in this section, i.e., SE_s, IE_s, SE_c, and IE_s.

Table 6.1 *Package-class* feature rating matrix combined with SE_c for the Web metamodel.

	Page	Static	Dynamic	Entity	Field
Web	1	1	1	0	0
Data	0	0	0	1	1

Table 6.2 *Class-structural feature* rating matrix combined with IE_s for the Web metamodel.

	title	meta	content	picture	list	entity	name	fields	isPK
Page	1	1	0	0	0	0	0	0	0
Static	1	1	1	1	0	0	0	0	0
Dynamic	1	1	0	0	1	1	0	0	0
Entity	0	0	0	0	0	0	1	1	0
Field	0	0	0	0	0	0	1	0	1

6.1.2.3 Data Encoder

Once package and class pairs as well as class and structural features have been extracted, MemoRec represents the relationships among them using two rating matrices to support class and structural feature recommendations. Given a metamodel, each row in the matrix corresponds to a package (class), and each column represents a class (structural feature). A cell is set to 1 if the package (class) in the corresponding row contains the class (structural feature) in the column, otherwise it is set to 0.

Table 6.1 and Table 6.2 illustrate how metamodel features are encoded into corresponding rating matrices. In particular, Table 6.1 shows the rating matrix combined with SE_c, whereas Table 6.2 reports the rating matrix combined with IE_s.

A 3D context-based ratings matrix is introduced to model the intrinsic relationships among various metamodels, package (classes) and class (structural feature). The third dimension of this matrix represents a metamodel, which is analogous to the so-called context in context-aware collaborative filtering systems. For example, Fig. 6.6 depicts three metamodels $M = (m_a, m_1, m_2)$ represented by three slices with four classes and five structural features: m_a is the *active metamodel* and it has an *active context* highlighted in dark gray. Both m_1 and m_2 are complete metamodels similar to m_a, and they are called *background data*, as they serve as a base for the recommendation process. On one hand, the more background metamodel we have, the better is the chance that we recommend relevant structural features. On the other hand, increasing the number of top similar metamodels will enlarge the ratings matrix, and thus will add more computational complexity.

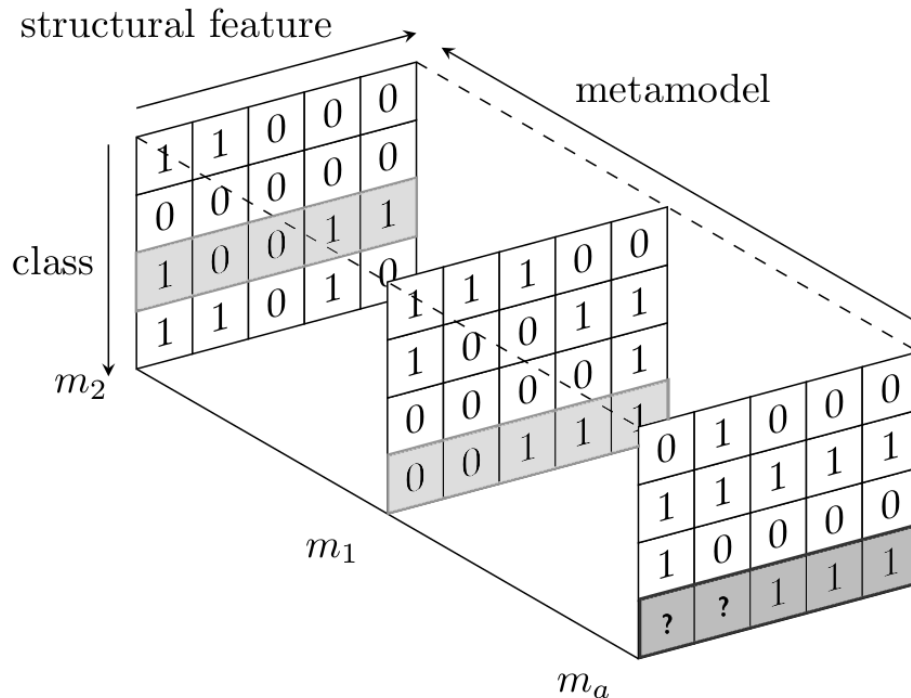


Fig. 6.6 Matrix representation of metamodels w.r.t. structural features and classes.

6.1.2.4 Similarity Calculator

The recommendation of suitable metamodel items, i.e., classes or structural features, is derived from similar metamodels and the active context, i.e., packages or classes. *Similarity Calculator* is a generic component, it can be used to compute similarity for both classes and structural features. Given an active context of a metamodel under development, it is essential to find the subset of the most similar ones, and then the most similar contexts in that set of metamodels. Based on the active context type, we create a weighted directed graph that models the relationships among metamodels and structural features to compute similarities. Moreover, we implemented a graph-based similarity function [66, 181] to calculate the similarities among metamodels.

In particular, we used two graph representations to support both class and structural feature recommendations. Each node in the graph represents either a metamodel or a structural feature. If metamodel m contains structural feature f, then there is a directed edge from m to f. The weight of the edge $m \rightarrow f$ corresponds to the number of times m includes f. Figure 6.7 depicts the graph for the set of projects in Fig. 6.6: white nodes represent structural features, blue nodes represent most similar metamodels to the input ones depicted in green. For instance, the *Web* metamodel has five classes and two of them define the attribute *name*. As a result, the edge *Web* $\rightarrow$ *name* contains a weight of 2. In the graph, a question mark

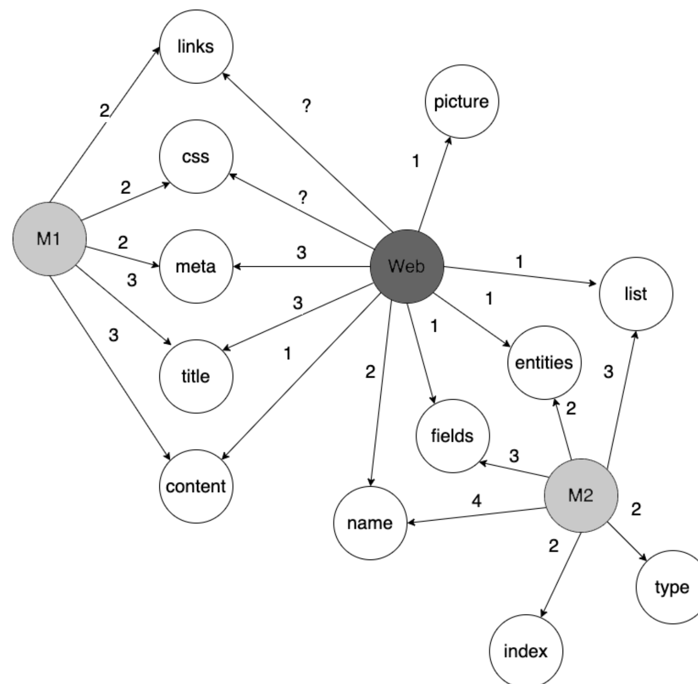


Fig. 6.7 Graph representation of metamodels and structural features.

represents missing information, i.e., for the active declaration in *Web*, we need to find out if invocations *links* and *css* shall be included or not.

Given the node representing a metamodel m, there are nodes connected to m via different edges, and they are called neighbor nodes. Considering $(n_1, n_2, .., n_l)$ as a set of neighbor nodes of m, the feature set of m is the vector $\phi = (\phi_1, \phi_2, .., \phi_l)$, where ϕ_k is the weight of node n_k, and computed using the *term-frequency inverse document frequency* function computed by the following formula:

$$\phi_k = f_{n_k} * log(\frac{|M|}{a_{n_k}}) \tag{6.1}$$

where f_{n_k} is the weight of the edge $m \rightarrow n_k$; $|M|$ is the number of all considered metamodels; and a_{n_k} is the number of metamodels connected to n_k.

The similarity between two metamodels m and n is comprehended as the cosine between their feature vectors $\phi = \{\phi_k\}_{k=1,..,l}$ and $\omega = \{\omega_j\}_{j=1,..,z}$, computed below:

$$sim_1(m,n) = \frac{\sum_{t=1}^{\pi} \phi_t \times \omega_t}{\sqrt{\sum_{t=1}^{\pi}(\phi_t)^2} \times \sqrt{\sum_{t=1}^{\pi}(\omega_t)^2}} \tag{6.2}$$

where π is the cardinality of the union of the sets of nodes by m and n.

Table 6.3 ϕ vectors for the metamodels depicted in Fig. 6.7.

	links	**css**	**media**	**title**	**content**	**picture**	**name**	**fields**	**entities**	**list**	**index**	**type**
Web	0	0	0.528	0.528	0	0	0	-0.301	-0.301	-0.301	0	0
M1	0	0	0	0.528	0.528	0	0	0	0	0	0	0
M2	0	0	0	0	0	0	1.204	0.528	0	0.528	0.602	0.602

Table 6.4 sim_1 matrix for the metamodels depicted in Fig. 6.7.

	Web	M1	M2
Web	1	**0.41**	-0.21
M1	**0.41**	1	0
M2	-0.21	0	1

Finally, the similarity between classes c and d is calculated with the Jaccard index given below:

$$sim_2(c,d) = \frac{|\mathbb{F}(c) \bigcap \mathbb{F}(d)|}{|\mathbb{F}(c) \bigcup \mathbb{F}(d)|} \tag{6.3}$$

where $\mathbb{F}(c)$ and $\mathbb{F}(d)$ are the sets of structural features for c and d, respectively.

By referring to the motivation example proposed in Section 6.1.1 and depicted in Fig. 6.7, we present a concrete application of the proposed formalisms. Table 6.3 reports the ϕ vectors for the *Web*, *M1*, and *M2* metamodels. Then, Table 6.4 lists the cosine similarity among vectors. Each cell reports the sim_1 score between the metamodels represented in the corresponding column and row. According to the results reported in Table 6.4, the *Web* metamodel is more similar to *M1* than to *M2*.

6.1.2.5 Recommendation Engine

This component is used to generate a ranked list of relevant items, i.e., classes and structural features that depend on the metamodel context, i.e., package and class. In the rest of this section, we present structural feature recommendations based on the class context. Analogously, we apply the same approach for recommending classes within packages.

Figure 6.6 depicts an instance of structural features rating matrices. In particular, the active metamodel m_a already includes three classes, and the modeler is working on the fourth class, corresponding to the last row of the matrix. The active class c_a contains two structural features, represented in the last two columns of the matrix, i.e., cells marked with 1. The first two cells are filled with a question mark (?), implying that at the time of consideration, it is not clear whether these two structural features should also be added into c_a. *Recommendation Engine* computes the missing ratings to predict additional structural features for the active class by exploiting the following collaborative filtering formula [49, 181]:

$$r_{c,f,m} = \overline{r}_c + \frac{\sum_{d \in topsim(c)} (R_{d,f,m} - \overline{r}_d) \cdot sim_2(c,d)}{\sum_{d \in topsim(c)} sim_2(c,d)} \tag{6.4}$$

Equation 6.4 is used to compute a score for the cell representing structural feature f, class c of metamodel m, where $topsim(c)$ is the set of top-N similar classes of c, $sim_2(c,d)$ is the similarity between two classes c and d, computed by Equation 6.3; $\overline{r}_c$ and $\overline{r}_d$ are calculated by averaging out all the ratings of c and d, respectively; $R_{d,f,m}$ is the combined rating of d for f in all the similar metamodels, computed in Equation 6.5 [49].

$$R_{d,f,m} = \frac{\sum_{n \in topsim(m)} r_{d,f,n} \cdot sim_1(m,n)}{\sum_{n \in topsim(m)} sim_1(m,n)} \tag{6.5}$$

where $topsim(m)$ is the set of top similar metamodels of m; and $sim_1(m,n)$ is the similarity between metamodels m and n, calculated by means of Equation 6.2. Equation 6.5 suggests that given the active metamodel, a more similar metamodel is assigned a higher weight. This makes sense in practice, since similar metamodels contain more relevant structural features than less similar metamodels. Using Equation 6.4 we compute all the missing ratings in the active class and get a ranked list of structural features with real scores in descending order. The list is then provided to modelers as recommendations.

6.1.3 Evaluation

This section describes the datasets and the process we conceived to evaluate the prediction performance of MemoRec. In particular, Section 6.1.3.1 presents the research questions to study the performance of our proposed approach. Section 6.1.3.2 gives an overview of the datasets used in the evaluation. Finally, the evaluation methodology is described in Section 6.1.3.3.

6.1.3.1 Research Questions

The following research questions are considered to investigate MemoRec's recommendation performance:

- **RQ**$_1$: *How well can MemoRec provide recommendations with different configurations?* We examine different configurations of MemoRec, i.e., the number of recommended items as well as the number of neighbor metamodels, to find the settings that bring the best performance.

- **RQ**$_2$: *How does the training data affect the performance of MemoRec?* We study the outputs of MemoRec by considering two different datasets to assess to what extent their quality can have ripple effects on the prediction accuracy of MemoRec;

- **RQ**$_3$: *How do the encoding schemes affect the performance of MemoRec?* Since the definition of a suitable encoding scheme is an iterative process, we compared refined versions of the encodings, i.e., IE_c and IE_s, with the initial ones, i.e., SE_c and SE_s to pin down which of them facilitates the best recommendation outcome for MemoRec.

6.1.3.2 Data Extraction

To evaluate the proposed approach, we exploited two independent datasets namely $\mathbf{D}_1$ and $\mathbf{D}_2$ as shown in Table 6.5, and they are described below.

- $\mathbf{D}_1$ is a **curated** dataset [300], which consists of 555 metamodels mined from GitHub and already labeled by humans. In particular, its metamodels have been already classified into nine categories, i.e., *Bibliography*, *Issue tracker*, *Project build*, *Review system*, *Database*, *Office tools*, *Petrinet*, *State machine*, and *Requirements specification*. Though MemoRec does not require the input data to be labeled, such predefined categories are beneficial to the recommendation as there is a high similarity among the metamodels within a category. This is important since MemoRec heavily relies on similarity to function (see Section 6.1.2), i.e., given a metamodel, it searches for relevant items from similar metamodels;

- $\mathbf{D}_2$ is a **raw and randomly collected** dataset using the GitHub API [4]. To aim for a reliable evaluation of MemoRec, we identified and filtered out from the dataset all the duplicated metamodels, resulting in a final set with 2,151 metamodels. By means of the GitHub API [4] we searched for files with the `.ecore` extension, which corresponds to Ecore metamodels. Due to the restrictions imposed by GitHub, e.g., it returns a maximum of 1,000 elements per query, we had to perform the searches by iteratively varying the query keywords. In particular, we used the `extension` qualifier as a base to search for `ecore` files. Then, we refined the query by adding typical `ecore` keywords, i.e., `ePackage`, `xml`, `eClass`, to name a few. Afterward, all the discovered metamodels were downloaded and collected in a dedicated folder. We removed all the files that we cannot directly parse with the EMF facilities [248] from the corpus of collected artifacts. Finally, we removed the duplicated metamodels by the following process: *(i)* a hash is computed for every collected `ecore` file based on its content; *(ii)* the obtained hashes are used to build a hashmap where the key is the hash itself, and the value is

Table 6.5 Datasets.

Number of Artifacts	$\mathbf{D}_1$	$\mathbf{D}_2$
Packages	669	3,140
Metaclasses	17,840	62,214
Structural features	31,688	159,323
Attributes	14,436	86,273
References	17,252	73,050

the corresponding file; *(iii)* if a duplicated key occurs in the map, we assume that the corresponding `ecore` is a duplicate and it is discarded.

6.1.3.3 Methodology

As described in Section 6.1.2, MemoRec can recommend classes or structural features, i.e., attributes and references, depending on the recommendation context, i.e., packages or classes. For this reason, we perform the experiments by exploiting both classes and structural feature recommendations. In the rest of this section, we use classes within packages and structural features within classes as the recommendation objective.

To study if MemoRec is applicable in real-world settings, we perform an offline evaluation by simulating the behavior of a modeler who partially defines a metamodel and needs pratical recommendations on how to do next. Figure 6.8 depicts the evaluation process with three consecutive steps, i.e., *Data Preparation*, *Recommendation Production*, and *Outcome Evaluation* explained as follows.

- **Data Preparation.** As seen in Fig. 6.8, starting from an input *Dataset*, we split it into two independent parts, i.e., *Training data* and *Testing data* (*Split ten-fold*). The former corresponds to the metamodels collected ex-ante, whereas the latter represents the metamodel being modeled, or the active metamodel. The *ten-fold* cross validation technique is used to conduct the evaluation as follows. The dataset is split into ten equal parts, one part represents the testing set and the remaining nine parts are combined to create a training set. We consider a modeler who is defining a metamodel *m*, so some parts of *m* are removed to mimic an actual metamodeling task: some packages/classes are already available in the active metamodel and the system should recommend additional packages/classes to be incorporated. For each metamodel in the *Testing data*, by the *Split input data* phase, a random package (class) that, together with the remaining packages/classes, is selected to be used as *Query data*. In the considered context, the first class/structural feature is kept as query data and all the others are

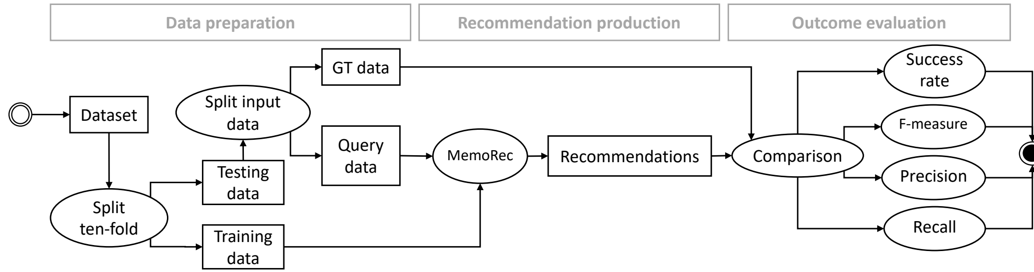


Fig. 6.8 Evaluation Process.

taken out to be used as ground-truth (GT) *data*. In other words, by the *Split input data* phase, package (class) is selected as the active context c. For c, only the selected classes are provided as query, while the rest is removed and saved as ground-truth data.

- **Recommendation production.** In this phase, the extracted *Query data* and *Training data* are fed as input for MemoRec, which in turn computes the final *Recommendations*. It is important to remark that the current version of MemoRec can recommend classes and structural features. The types of the recommended attributes and relationships are not supported yet. This represents our next step to further develop the proposed approach.

- **Outcome evaluation.** The performance of MemoRec is measured by comparing the recommendation outcomes with the ground-truth data (*GT data*), exploiting the quality metrics, i.e., *Success rate*, *Precision*, and *Recall*. Furthermore, we measure the time needed to perform a prediction, by using a laptop with 2,7 GHz Intel Core i7 quad-core 16GB RAM, and macOS Catalina 10.15.5.

6.1.4 Results

RQ$_1$: *How well can MemoRec provide recommendations with different configurations?* We conducted experiments on the curated dataset (i.e., $\mathbf{D}_1$) by varying the number of neighbour nodes k of the input metamodel, i.e., $k = \{1,5,10,15,20\}$, and the value of N, i.e., $N = \{1,10,20\}$. The rationale behind the selection of those values is as follows. First, we should not present a long list of recommended items since it may confuse the modeler, thus we select $N = 20$ as the maximum value. Second, since the number of neighborhood items impacts on the computational complexity (cf. Equation (6.5)), it is impractical to use a large number of metamodels as neighbors. Therefore, we consider the following values $k = \{1,5,10,15,20\}$. In the experiments, we use SE$_c$ and SE$_s$ as encoding schemes to

compute recommendations for classes and structural features, respectively (the results related to the adoption of the other encoding schemes are presented in the discussion of RQ_3).

Table 6.6 and Table 6.7 show the average success rates obtained by running the ten-fold cross-validation technique to recommend structural features and classes, respectively by using different cut-off values N. In particular, Table 6.6 depicts the success rate obtained for recommending structural features classes. According to Table 6.6, we can observe that using more neighbour metamodels to compute recommendations brings a better success rate when the first recommendation item is considered. For instance, MemoRec gets a success rate@1 of 0.153 and 0.202 when k=1 and k=20, respectively. This is not confirmed when a longer list of recommendation items is considered, i.e., $N = \{10,20\}$. Moreover, MemoRec yields a better performance when we increase the cut-off value N. Take as an example, for k=20, N=1, the obtained success rate is 0.202 which is less than a half of 0.479, the corresponding value when N=20. A longer list of recommended items means an increase in the match rate, however the modeler may tire of skimming through it. Thus, in practice, we should choose a suitable cut-off value N.

Table 6.6 Success rate for structural feature recommendations, $k = \{1,5,10,15,20\}$, by considering the $\mathbf{D}_1$ dataset.

k	**SR@1**	**SR@10**	**SR@20**
1	0.152	0.394	0.501
5	0.181	0.406	0.488
10	0.202	0.419	0.502
15	0.200	0.392	0.494
20	0.202	0.362	0.479

Table 6.7 report the success rate obtained with class recommendations. Partly similar to the results presented for recommending structural features classes in Table 6.6, we see that incorporating more neighbors to compute recommendations is useful for a small k, i.e., $k = \{1,5,10\}$. However, starting from k=15, there is a decrease in success rate by all the cut-off values N. We suppose that this happens due to the adoption of new neighbors that introduces only noise.

Answer to $\mathbf{RQ}_1$. Considering a certain number of similar metamodels contributes to more relevant recommendations. Using data encoded with the SE_c and SE_s encoding schemes allows MemoRec to predict better classes within a package than structural features within a class. Moreover, by considering a longer list of recommended items, MemoRec obtains an increase in success rate.

Table 6.7 Success rate for class recommendations, $k = \{1, 5, 10, 15, 20\}$, by considering the $\mathbf{D}_1$ dataset.

k	**SR@1**	**SR@10**	**SR@20**
1	0.285	0.468	0.487
5	0.204	0.617	0.690
10	0.191	0.613	0.702
15	0.215	0.583	0.694
20	0.187	0.539	0.661

RQ$_2$: ***How does the training data affect the performance of MemoRec?*** We conducted similar experiments previously presented by measuring also the performance induced by the adoption of the dataset $\mathbf{D}_2$. As previously described in Section 6.1.3, $\mathbf{D}_1$ and $\mathbf{D}_2$ are different in terms of size and quality. In particular, $\mathbf{D}_1$ contains different groups of similar metamodels. Each group is labeled to refer the application domain that the metamodels in the considered group are intended to describe. Thus, as done for answering RQ$_1$, we performed experiments by varying the number of neighbour nodes of the input metamodel and the value of N.

Table 6.8a and Table 6.8b show the average success rates obtained by running the ten-fold cross-validation technique to recommend structural features and classes, respectively by using different cut-off values N.

According to Table 6.8a, it is evident that using more neighbour metamodels to compute recommendations brings a better success rate for both datasets when the first recommended item is considered. An increasing number of neighbor k does not improve the success rate values when a longer list of recommendations is considered, i.e., $SR@10$ and $SR@20$. However, by using the randomly created dataset $\mathbf{D}_2$ success rate is lower than that of $\mathbf{D}_1$. For instance, with $\mathbf{D}_1$, MemoRec gets a success rate@1 of 0.159 and 0.202 when k=1 and k=20, respectively, whereas with $\mathbf{D}_2$ the corresponding values are 0.114 and 0.161. The same trend can also be seen with other cut-off values. As in the case of $\mathbf{D}_2$, MemoRec yields a better performance when we increase the cut-off value N. Take as an example, with $\mathbf{D}_2$ and k=20, N=1, the corresponding success rate is 0.178 which is less than a half of 0.373, the corresponding value when N=20. In any case, the success rate related to the adoption of $\mathbf{D}_2$ is always lower than that of $\mathbf{D}_1$.

Table 6.8b report the success rate obtained with class recommendations by comparing the adoption of $\mathbf{D}_1$ and $\mathbf{D}_2$. The decrease in accuracy related to the adoption of $\mathbf{D}_2$ as shown in Table 6.8a is confirmed also in Table 6.8b. To further study MemoRec's performance, we compute and report in Table. 6.9, Table 6.10, and Table 6.11 the precision, recall, and f-measure $\mathbf{D}_1$ and $\mathbf{D}_2$. For this setting, the number of recommended items N was varied from

Table 6.8 Success rate, $k = \{1, 5, 10, 15, 20\}$, by comparing the adoption of $\mathbf{D}_1$ and $\mathbf{D}_2$

k	SR@1		SR@10		SR@20	
	$\mathbf{D}_1$	$\mathbf{D}_2$	$\mathbf{D}_1$	$\mathbf{D}_2$	$\mathbf{D}_1$	$\mathbf{D}_2$
1	0.153	0.114	0.394	0.328	0.502	0.397
5	0.181	0.150	0.406	0.363	0.489	0.452
10	0.202	0.161	0.419	0.340	0.501	0.413
15	0.200	0.170	0.392	0.327	0.494	0.395
20	0.202	0.178	0.362	0.320	0.479	0.373

(a) for structural feature recommendations

k	SR@1		SR@10		SR@20	
	$\mathbf{D}_1$	$\mathbf{D}_2$	$\mathbf{D}_1$	$\mathbf{D}_2$	$\mathbf{D}_1$	$\mathbf{D}_2$
1	0.285	0.147	0.468	0.307	0.487	0.331
5	0.204	0.173	0.617	0.493	0.691	0.589
10	0.191	0.150	0.613	0.445	0.702	0.571
15	0.215	0.147	0.583	0.397	0.694	0.521
20	0.187	0.141	0.539	0.362	0.661	0.478

(b) for class recommendations

Table 6.9 Precision values

N	$K=1$	$K=5$	$K=10$	$K=15$	$K=20$
1	0.155	0.187	0.202	0.208	0.208
2	0.157	0.186	0.191	0.193	0.192
3	0.139	0.160	0.172	0.170	0.167
4	0.125	0.142	0.146	0.148	0.144
5	0.116	0.126	0.126	0.129	0.126
6	0.106	0.115	0.114	0.116	0.114
7	0.097	0.105	0.105	0.106	0.102
8	0.089	0.093	0.095	0.096	0.094
9	0.084	0.085	0.086	0.087	0.084
10	0.079	0.080	0.081	0.083	0.077

(a) for structural feature recommendations using $\mathbf{D}_1$

N	$K=1$	$K=5$	$K=10$	$K=15$	$K=20$
1	0.114	0.150	0.161	0.170	0.178
2	0.107	0.139	0.147	0.146	0.152
3	0.096	0.117	0.120	0.118	0.121
4	0.087	0.103	0.104	0.101	0.102
5	0.078	0.092	0.092	0.089	0.091
6	0.074	0.085	0.083	0.081	0.081
7	0.069	0.078	0.076	0.074	0.074
8	0.064	0.072	0.070	0.068	0.069
9	0.060	0.068	0.066	0.064	0.063
10	0.057	0.063	0.061	0.059	0.059

(b) for structural feature recommendations using $\mathbf{D}_2$

N	$K=1$	$K=5$	$K=10$	$K=15$	$K=20$
1	0.236	0.193	0.204	0.198	0.180
2	0.235	0.213	0.179	0.182	0.164
3	0.218	0.198	0.162	0.159	0.142
4	0.209	0.181	0.156	0.149	0.138
5	0.196	0.175	0.151	0.144	0.124
6	0.185	0.168	0.150	0.134	0.114
7	0.177	0.160	0.147	0.131	0.112
8	0.166	0.148	0.142	0.130	0.114
9	0.163	0.151	0.144	0.127	0.110
10	0.156	0.158	0.146	0.125	0.105

(c) for class recommendations using $\mathbf{D}_1$

N	$K=1$	$K=5$	$K=10$	$K=15$	$K=20$
1	0.148	0.173	0.147	0.144	0.136
2	0.138	0.165	0.152	0.142	0.132
3	0.132	0.154	0.139	0.129	0.119
4	0.123	0.144	0.127	0.118	0.107
5	0.116	0.135	0.116	0.107	0.096
6	0.111	0.128	0.110	0.101	0.090
7	0.107	0.122	0.107	0.096	0.085
8	0.103	0.117	0.102	0.093	0.081
9	0.099	0.113	0.099	0.090	0.079
10	0.096	0.112	0.096	0.087	0.076

(d) for class recommendations using $\mathbf{D}_2$

1 to 10, attempting to examine the performance for a considerably long list of items. The value of k was varied of 1 to 20 with 5 as the step, considering a large number of neighbors.

Table. 6.9 and Table. 6.10 confirm that by considering a curated dataset with more similar metamodels, MemoRec has better prediction performance than using a raw dataset. Moreover, MemoRec reaches better prediction performance in recommending structural features classes than classes packages.

Answer to RQ$_2$. The quality of the input data plays a key role in MemoRec's performance. Curated datasets with more similar metamodels allow MemoRec to improve its prediction performance, even if the size of such datasets is smaller than that of those randomly collected.

Table 6.10 Recall values

N	$K=1$	$K=5$	$K=10$	$K=15$	$K=20$
1	0.054	0.063	0.070	0.070	0.071
2	0.108	0.128	0.128	0.128	0.128
3	0.142	0.161	0.174	0.168	0.163
4	0.163	0.188	0.192	0.191	0.184
5	0.182	0.203	0.202	0.204	0.195
6	0.195	0.216	0.214	0.217	0.209
7	0.205	0.227	0.225	0.225	0.214
8	0.214	0.228	0.235	0.233	0.224
9	0.228	0.233	0.239	0.239	0.226
10	0.237	0.242	0.248	0.250	0.228

(a) for structural feature recommendations using $\mathbf{D}_1$

N	$K=1$	$K=5$	$K=10$	$K=15$	$K=20$
1	0.037	0.051	0.057	0.060	0.063
2	0.070	0.095	0.103	0.102	0.107
3	0.092	0.115	0.123	0.120	0.124
4	0.108	0.133	0.139	0.134	0.136
5	0.121	0.145	0.150	0.145	0.148
6	0.135	0.159	0.160	0.154	0.155
7	0.146	0.169	0.168	0.165	0.164
8	0.153	0.177	0.177	0.171	0.172
9	0.162	0.188	0.185	0.178	0.177
10	0.169	0.194	0.190	0.183	0.182

(b) for structural feature recommendations using $\mathbf{D}_2$

N	$K=1$	$K=5$	$K=10$	$K=15$	$K=20$
1	0.032	0.039	0.048	0.049	0.048
2	0.056	0.074	0.084	0.088	0.088
3	0.067	0.091	0.102	0.111	0.109
4	0.077	0.099	0.112	0.119	0.118
5	0.083	0.109	0.120	0.132	0.127
6	0.090	0.118	0.132	0.138	0.134
7	0.099	0.124	0.143	0.149	0.144
8	0.103	0.127	0.152	0.157	0.152
9	0.109	0.137	0.162	0.163	0.158
10	0.114	0.149	0.172	0.169	0.162

(c) for class recommendations using $\mathbf{D}_1$

N	$K=1$	$K=5$	$K=10$	$K=15$	$K=20$
1	0.018	0.030	0.030	0.032	0.032
2	0.034	0.058	0.062	0.063	0.061
3	0.044	0.075	0.078	0.078	0.077
4	0.050	0.085	0.087	0.085	0.083
5	0.054	0.092	0.092	0.090	0.087
6	0.058	0.099	0.098	0.096	0.092
7	0.062	0.105	0.106	0.101	0.095
8	0.065	0.110	0.111	0.106	0.099
9	0.069	0.116	0.116	0.112	0.105
10	0.072	0.123	0.121	0.116	0.109

(d) for class recommendations using $\mathbf{D}_2$

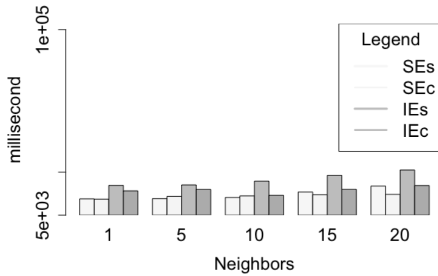


Fig. 6.9 Dataset $\mathbf{D}_1$

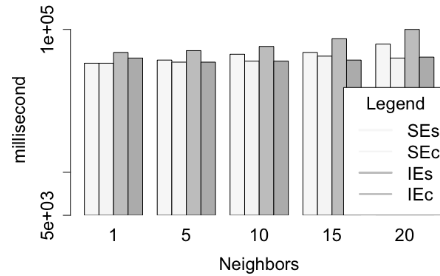


Fig. 6.10 Dataset $\mathbf{D}_2$

Fig. 6.11 Average execution time.

RQ$_3$: ***How do the encoding schemes affect the performance of MemoRec?*** Before addressing this research question, we discuss the recommendations produced by MemoRec on the example shown in Fig. 6.12. More specifically, MemoRec has been applied on the `Bibtex` metamodel where the modeler is asking for the recommendation for two different contexts, i.e., the `Article` class (rounded in red) and the `Bibtex` package (rounded in blue). Because of the different context's nature, we show the recommendations to `Article` class by considering the IE$_s$ and SE$_s$ encodings. In contrast, the IE$_c$ and SE$_c$ encodings are discussed by considering the `Bibtex` package as the active context. We briefly summarize the four

Table 6.11 F-measure values

N	$K=1$	$K=5$	$K=10$	$K=15$	$K=20$
1	0.080	0.094	0.104	0.105	0.106
2	0.128	0.152	0.153	0.154	0.154
3	0.140	0.160	0.173	0.169	0.165
4	0.141	0.162	0.166	0.167	0.162
5	0.142	0.155	0.155	0.158	0.153
6	0.137	0.150	0.149	0.151	0.148
7	0.132	0.144	0.143	0.144	0.138
8	0.126	0.132	0.135	0.136	0.132
9	0.123	0.125	0.126	0.128	0.122
10	0.119	0.120	0.122	0.125	0.115

(a) for structural feature recommendations using $\mathbf{D}_1$

N	$K=1$	$K=5$	$K=10$	$K=15$	$K=20$
1	0.056	0.076	0.084	0.089	0.093
2	0.085	0.113	0.121	0.120	0.126
3	0.094	0.116	0.121	0.119	0.122
4	0.096	0.116	0.119	0.115	0.117
5	0.095	0.113	0.114	0.110	0.113
6	0.096	0.111	0.109	0.106	0.106
7	0.094	0.107	0.105	0.102	0.102
8	0.090	0.102	0.100	0.097	0.098
9	0.088	0.100	0.097	0.094	0.093
10	0.085	0.095	0.092	0.089	0.089

(b) for structural feature recommendations using $\mathbf{D}_2$

N	$K=1$	$K=5$	$K=10$	$K=15$	$K=20$
1	0.056	0.065	0.078	0.079	0.076
2	0.090	0.110	0.114	0.119	0.115
3	0.102	0.125	0.125	0.131	0.123
4	0.113	0.128	0.130	0.132	0.127
5	0.117	0.134	0.134	0.138	0.125
6	0.121	0.139	0.140	0.136	0.123
7	0.127	0.140	0.145	0.139	0.126
8	0.127	0.137	0.147	0.142	0.130
9	0.131	0.144	0.152	0.143	0.130
10	0.132	0.153	0.158	0.144	0.127

(c) for class recommendations using $\mathbf{D}_1$

N	$K=1$	$K=5$	$K=10$	$K=15$	$K=20$
1	0.032	0.051	0.050	0.052	0.052
2	0.055	0.086	0.088	0.087	0.083
3	0.066	0.101	0.100	0.097	0.094
4	0.071	0.107	0.103	0.099	0.093
5	0.074	0.109	0.103	0.098	0.091
6	0.076	0.112	0.104	0.098	0.091
7	0.079	0.113	0.106	0.098	0.090
8	0.080	0.113	0.106	0.099	0.089
9	0.081	0.114	0.107	0.100	0.090
10	0.082	0.117	0.107	0.099	0.090

(d) for class recommendations using $\mathbf{D}_2$

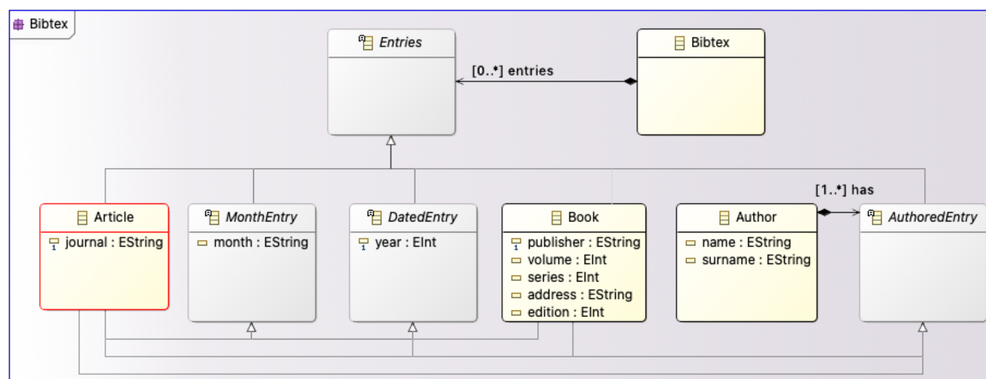


Fig. 6.12 The `Bibtex` metamodel.

proposed schemes in Section 6.1.2 as follows. SE_s consists of *class-structural feature* pairs for each structural feature directly defined within a class while IE_s also includes structural feature inherited from the superclasses; IE_c consists of *package-class* pairs while SE_c flattens packages within a default artificial package.

Table 6.12 shows the list of top-10 recommended items for each encoding scheme, i.e., SE_c, IE_c, SE_s, and SE_c. As described in Section 6.1.2, the type of the active context discriminates which encoding MemoRec needs to adopt. By observing the outcomes of Table 6.12, it can be seen that using IE_s and SE_s produces diverse recommended items.

Table 6.12 Predicted recommendations for the `Article` metaclass and `Bibtex` package of the metamodel in Fig. 6.12.

	IE_s	SE_s
`Article` metaclass	containerPage	address
	id	edition
	toolspecifics	journal
	abstractPrefix	month
	abstractText	note
	acceptDirty	number
	actor	pages
	address	publisher
	attachment	series
	authorName	volume
	IE_c	SE_c
`Bibtex` package	AbstractCondition	Abstract
	Assign	CommitteeMember
	AssignExtra	AcademiaOrganization
	AttributeCondition	AcademicEvent
	Book	Academic_Institution
	ConditionalState	AcceptRating
	ConditionalTransition	Acceptance
	Editor	AcceptedPaper
	EmphasisValue	AccommodationPlace
	Event	Account

The result provided by IE_s seems to contain more generic recommendations, e.g., id, actor, attachment, to name a few, while SE_s provides more focused recommended structural features. This intuition is confirmed by manually analyzing significant recommendation examples. Though both IE_s and SE_s provide useful suggestions to the modeler, the augmented information of SE_c allows MemoRec to predict more specific structural features. Referring to the scores obtained with IE_c and SE_c, we conclude that the results depend on the similarities among the training metamodels. By flattening package structure, SE_c mainly uses class names to compute similarities, while IE_c includes also package structure. So, when a similar metamodel is identified, SE_c can recommend classes that belong to any package.

To answer RQ_3, we conducted experiments on $\mathbf{D}_1$ by comparing pairwise the encoding schemes, i.e., SE_s versus IE_s and SE_c versus IE_c. As we mentioned before, we should select a suitable number of neighbor metamodels to compute recommendations to maintain a trade-off between efficiency and effectiveness. Thus, to simplify the evaluation, we set the

number of neighbors k to 5.[6] The evaluation metrics computed by the encoding schemes for structural features and classes are reported in Table 6.13a and Table 6.13b, respectively.

Table 6.13a demonstrates an evident outcome: by using IE_s as the encoding scheme, we obtain a better prediction performance than that when using SE_s. For instance, given $N = 1$, we get a success rate of 0.241 and 0.181 for IE_s and SE_s encodings, respectively. Similarly by other evaluation metrics, i.e., precision and recall, IE_s helps achieve a superior performance. When we consider larger cut-off values, i.e., $N = \{5, 10, 15, 20\}$, IE_s is always beneficial to the recommendation outcomes, as it brings in higher quality indicators. Take as an example, with N=20, using IE_s yields a success rate of 0.604, which is much higher than 0.489, the corresponding value for SE_s.

Next, we analyze the recommendations by using SE_c and IE_c as shown in Table 6.13b. For success rate and precision, using IE_c help MemoRec perform better than using SE_c. However, IE_c negatively impacts on the recall values. In our opinion, it is due to the flattening operation which affects the similarity function, making metamodels too similar. In this case, the recommender engine is limited to suggests the most common classes. For instance, because of metamodelling best-practice, `NamedElement` is commonly used in a metamodel. This impacts on success rate and precision, but recall goes down because the recommended items do not depend on the metamodel context.

Through the experiment, we see that a suitable encoding scheme fosters better prediction performance. In this respect, we believe that the introduction of Natural Language Processing (NLP) steps, i.e., stemming, lemmatization, and stop words removal, can boost up the accuracy of MemoRec. In particular, preprocessing steps can be employed to reduce the usage of different terms with very close semantics and, thus, to increase the corresponding term usages. For instance, the word "*reference*" and its plural form "*references*" are two conjugations of the same noun. Even though the two words have the same semantics, currently, MemoRec does not match those two terms and considers them different by negatively affecting the resulting MemoRec performance. We consider the integration of NLP techniques as our future work.

The average execution time among ten folds on various values of k for both datasets is depicted in Fig. 6.9 and Fig. 6.10. It can be seen that while IE_s helps MemoRec achieve a good prediction performance, it sustains a high computational complexity, resulting in prolonged execution time. This is understandable since compared to other techniques, the encoding scheme incorporates more information from metamodels for its computation.

[6] For each dataset and N value, the average success rate is higher with k= 5 than that of other values of k.

Table 6.13 Success rate, precision and recall for class recommendations

N	SR@N		Precision@N		Recall@N	
	IE_s	SE_s	IE_s	SE_s	IE_s	SE_s
1	24.074	18.113	0.241	0.181	0.078	0.061
5	39.074	33.207	0.119	0.121	0.169	0.196
10	48.333	40.566	0.084	0.078	0.241	0.245
15	55.926	46.603	0.068	0.059	0.291	0.271
20	60.370	48.867	0.058	0.047	0.331	0.284

(a) using SE_s and IE_s encodings.

N	SR@N		Precision@N		Recall@N	
	IE_c	SE_c	IE_c	SE_c	IE_c	SE_c
1	22.593	20.370	0.226	0.204	0.014	0.037
5	56.296	53.333	0.220	0.196	0.061	0.117
10	65.926	61.667	0.194	0.163	0.100	0.152
15	70.370	66.667	0.186	0.158	0.139	0.189
20	73.333	69.074	0.167	0.140	0.159	0.210

(b) using SE_c and IE_c encodings.

Answer to RQ_3. Inherited structural features (IE_s) enable MemoRec to achieve a superior performance compared to using SE_s, despite a higher computational complexity. With IE_s, MemoRec predicts better structural features within a class than classes within a package.

6.1.5 Threats to validity

In this section we give a discussion of threats, which might harm the validity of the performed experiments. In particular, we discuss threats with respect to internal and external validity as follows.

Internal validity. Such threats refer to internal factors that might affect the outcomes of the performed experiments. A possible threat is represented by the datasets that have been used for the experiments. We mitigated such a threat by using two completely different datasets, and one of them has been randomly created without performing any data curation activities. Another internal threat to validity factor is represented by the adopted encoding schemes. Also in this case, we mitigated the issue by employing different encoding schemes. However, by considering data and encoding dimensions, we managed to identify distinctive characteristics of the approach that resulted to be valid independently from the adopted encoding schemes and input data sets, i.e., graph builder, similarity calculator, and recommendation engine.

External validity. It is related to factors that can affect the generalizability of our findings, by possibly making the obtained results not valid outside the scope of this study. We mitigated the issue by evaluating MemoRec in different scenarios, with the aim of simulating several usages of the systems, e.g., by varying the number of neighbour metamodels, and the size of the list of recommended items. Another threat to validity can be the fact that currently, we do not consider the sequences of actions that are operated to lead to a given metamodel. We believe that alternative approaches like LSTM (Long Short-Term Memory) [110] can be a possible candidate to produce recommendations that rely on creation sequences. Moreover, it is crucial to investigate how modelers perceive MemoRec. In this respect, we plan to conduct a user study where human evaluators are asked to give their assessment for recommendations provided by MemoRec.

6.2 MORGAN

6.2.1 Graph kernel similarity

A graph is often represented as an adjacency matrix **A** of size $N \times N$, where N is the number of nodes. If each node is characterized by a set of M features, then a dimension of feature matrix X is $N \times M$. Graph-structured data is complex, and thus it brings a lot of challenges for existing machine learning algorithms.

Given a graph, the prediction phase can be realized by following two different strategies, i.e., *link-level prediction* or *node-level classification.* The former requires to represent the relationship among nodes in graph and predict if there is a connection between two entities. In latter, the task is to understand node embedding for every node in a graph by looking at the labels of their neighbours. With *graph-level classification*, the entire graph needs to be classified into suitable categories.

Different kinds of algorithms can be used to produce recommendations, i.e., vector-space embedding [216] and graph kernel [274]. The former involves the annotation of nodes and edges with vectors that describe a set of features. Thus, the prediction capabilities of the model strongly depend on encoding salient characteristics into numerical vectors. The main limitation is that the vectors must be of a fixed size, which negatively impacts the handling of more complex data. In contrast, the latter works mostly on graph structure by considering three main concepts, i.e., paths, subgraphs, and subtrees. Such techniques support feature embeddings with mutable sizes, which eventually may lead to interesting results in the modeling domain [52].

Formally, a graph kernel is a symmetric positive semidefinite function k defined for a pair G_i and G_j such that the following equation is satisfied:

$$k(G_i, G_j) = \langle \phi(G_i), \phi(G_j) \rangle_{\mathscr{H}} \tag{6.6}$$

where $\langle \cdot, \cdot \rangle_{\mathscr{H}}$ is the inner product defined into a Hilbert Space H.

A graph kernel computes the similarity between two graphs by adopting several different strategies. Among these, the Weisfeiler-Lehman optimal assignment algorithm (WLOA kernel hereafter) [136] is built on top of existing graph kernels and it is inspired by the Weisfeiler-Lehman test of graph isomorphism [282].

In the previous version of MORGAN [68], we opted for the WLOA technique to measure the similarity as it offers a linear complexity. The algorithm replaces each vertex with a multiset representation consisting of the original one plus neighbors' features. Given two

graphs *G* and *G'*, the WLOA kernel is defined as follows:

$$k(G,G') = K^k_{\mathfrak{B}}(V,V') \tag{6.7}$$

where k is the employed base kernel, defined below:

$$k(v,v') = \sum_{i=0}^{h} \delta(\tau_i(v), \tau_i(v')) \tag{6.8}$$

where $\tau_i(v)$ is the label of node v at the end of ith iteration of the Weisfeiler-Lehman algorithm. However, for this kernel recommendation time is strongly bounded by the nature of the input data.

In this extension, we consider a different class of kernel function, namely the Vertex Histogram [251], to analyze to what extent the adoption of a different technique can reduce the whole prediction time.

In practice, given a pair of graphs G and G', let f, f' be the vertex histograms of the two graphs. The Vertex Histogram kernel is computed according to the following formula:

$$k(G,G') = \langle \mathbf{f}, \mathbf{f'} \rangle \tag{6.9}$$

where $\mathbf{f}, \mathbf{f'}$ are the vertex label histograms obtained by applying the mapping function $\ell : \mathscr{V} \rightarrow \mathscr{L}$ that assigns labels to the vertices of the graphs.[7]

Even though the underlying structure is not considered by the employed kernel, our empirical results demonstrate that MORGAN achieves better performances in terms of accuracy and time computation compared to the WeisfeilerLehman for this specific task. Therefore, we employ such a strategy to compute the similarity among the considered models and metamodels, which are encoded as graphs as described in the next section.

6.2.2 MORGAN architecture

This section describes in detail the proposed approach, whose architecture is depicted in Fig. 6.13. Starting from a partial model, MORGAN uses tailored model parsers to extract relevant information in textual data format. Alongside the two independent parsers developed in the original work, i.e., one for metamodels and the other for models, we conceive an additional parser tailored for JSON schema. The NLP module executes the encoding to generate graphs that can be used by the underlying graph kernel engine. In this work, we enrich this component with a lemmatization strategy to increase the amount of valuable information.

[7]This process is internally performed by the Python implementation offered by Grakel.

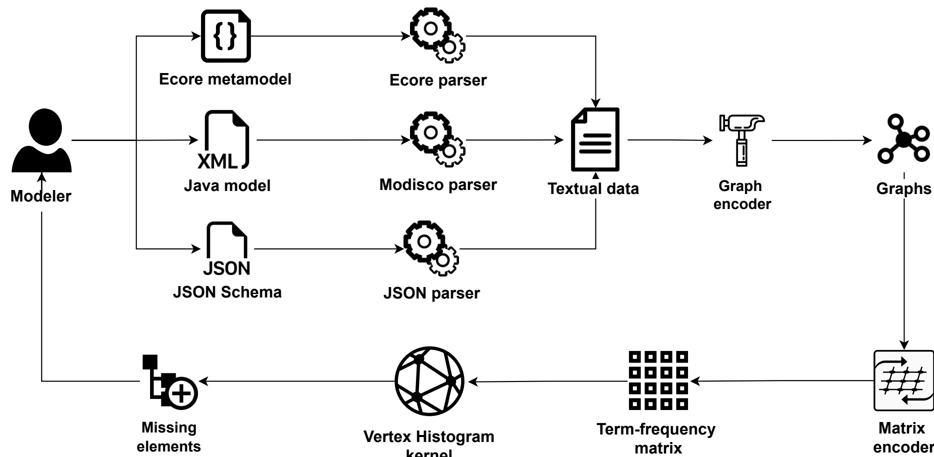


Fig. 6.13 The MORGAN architecture.

Furthermore, we adopted the Vertex Histogram kernel as the underpinning recommendation engine which is based on item frequency. As the final recommendations, MORGAN returns the missing elements of the artifact under development, i.e., either a metamodel or a model.[8] We describe the constituent components in the succeeding subsections.

6.2.2.1 Parsers

This component extracts relevant information from the input artifact by using four different parsers, i.e., *Ecore parser* for metamodels, *MoDisco parser* for MoDisco models, *UML parser* for UML models, and a standard *JSON parser*[9] for JSON schema. Because metamodels and models are expressed in the XMI or Ecore format, the first three parsers rely on EMF utilities[10] to extract their information. In contrast, the JSON schema parsers rely on utilities that embody the standard reference schema.

It is worth noting that this component is compliant with a well-structured encoding scheme. In such a way, the proposed tailored parsers follow the same structure during the data gathering phase, i.e., they are agnostic from the underlying model artifact. The generic encoding scheme is defined as follows:

$$instance\ id, (\langle relationship\ tuple \rangle)^* \tag{6.10}$$

[8] To facilitate the presentation, the common name *artifact* refers to all kinds of considered items, namely *metamodel*, *model*, and *JSON schema*, unless otherwise explicitly stated.

[9] https://github.com/google/gson

[10] https://www.eclipse.org/modeling/emf/

where the instances are represented by their identifier (e.g., name) and the relationships are tuples that encode related concepts depending on the notation. It is worth noting that a tuple can be composed of two or three elements, according to the considered modeling artifact.

▷ **Metamodel parsing.** Starting from a metamodel, *Ecore parser* excerpts the list of metaclasses and their structural features, i.e., attributes and references. In particular, the metaclass name identifies the corresponding metaclass instance while each relationship is represented as a *triple* defined as follows:

$$\textit{metaclass-name}\ (\langle Name, Type, [reference \mid attribute]\rangle)^* \tag{6.11}$$

where *Name* is the name of the structural feature, *Type* characterizes the type element, e.g., EString or EInt, and *Relation* identifies the type of relation between the element and the class, i.e., attribute or reference. Following the aforementioned scheme, the *Actor* metaclass described in Fig. 6.1a is encoded as `Actor (name, EString, attribute)`. These elements improve MORGAN's recommendation capabilities as they enable the modeler to add further information to the partial model.

▷ **MoDisco model parsing.** *MoDisco parser* is used to extract valuable data from models. In particular, similar to *Ecore parser*, *MoDisco parser* explores *xmi* trees to elicit valuable elements, i.e., each MoDisco model is represented as a list of *Java classes* followed by *method definition*s and *field declaration*s. The class name represents the instance identifier while each relationship is represented as a *triple* defined as follows:

$$\textit{class-name}\ (\langle Name, Return\ type, [method \mid field]\rangle)^* \tag{6.12}$$

where *Name* is the method or field name, *ReturnType* is the method or field type, and *Relation* identifies the type of relation between the class members, i.e., method or field, and the class. For instance, the signature of the getError method belonging to *LateBinding* class depicted in Figure 6.2a is translated as `LateBinding (getError, EString, method)`.

▷ **UML model parsing.** To extract the same type of information from models belonging to ModelSet, a UML parser has been conceived by exploiting EMF utilities. It is worth noting that the structure of the parser is similar to the previous ones, i.e., it navigates the tree structure of the UML models given as input. To be consistent with the parser requirements, we elicit *properties* and *operations* for each UML class. The encoding scheme for a single UML class is the following:

$$\textit{class-name}\ (\langle Name, Type\rangle)^* \tag{6.13}$$

where *Name* is the name of the property or operation of the class while *Type* can be *operation* or *property*. For instance, a UML class named *Teacher* with the property *name* and the operation *setMark* is translated into `Teacher (name, property) (setMark,operation)`.

▷ **JSON schema parsing.** Similarly to the metamodels that provide the notation to express models, JSON schema [5] is a dedicated language to define and validate JSON documents. A JSON schema document consists of two main types of schemas, i.e., *BooleanSchema* and *ObjectSchema*. In the scope of this paper, we focus on the latter that defines the hierarchical structure of the schema, including *(i)* the *Type* that could be a *SimpleType* or *object*, and *(ii)* a list of *Keyword* elements that defines the properties of the object. Each element of this list contains a *KeySchemaPair* that explicitly defines the structural features of an object or the primitive type. In this case, we opt for *ObjectSchema* name as the instance identifier, whereas the list of properties identified by *KeywordsPair* is used to identify the relationship tuples. We encoded such data as following:

$$\textit{ObjectSchema-name}\ (\langle name, [object \mid int \mid string \mid ...]\rangle)^* \tag{6.14}$$

If *Type* is a primitive type, e.g., string or boolean, *Keywords* are mapped to attributes, otherwise they are contained objects. In such a way, the JSON schema parser is used to extract pairs consisting of *ObjectSchema* and contained properties. This parser is capable of extracting the same type of information compared to the aforementioned ones. An excerpt of the encoding of *locations* highlighted in Fig. 6.3a can be represented as `Locations (name, string)(longitude, int)(mentions, object)`.

6.2.2.2 Graph Encoder

The next step is to build graphs from textual files produced by the parsing phase. In the former version of the tool [68], the artifact *Encoder* component applied a standard NLP pipeline composed of three main steps, i.e., stop-words removal, string cleaning, and stemming. We employ the lemmatization strategy instead of stemming by adopting the well-founded algorithm based on WordNet[11] The rationale behind this choice is that the former strategy is a process for removing the commoner morphological and inflexional endings from English words [205]. However, this could lead to incorrect meaning and spelling since the semantic is not considered at all. Meanwhile, *lemmatization* considers the context and extracts the lemma of each word, i.e., its base form. To enable this process, a large thesaurus of English terms is needed since the lemmatizer component involves the morphological analysis of each word. In the scope of the presented work, we make use of the WordNet Lemmatizer utility provided by

[11] https://wordnet.princeton.edu/

the *nltk* Python library[12] to extract the *lemma* from each analyzed term. Afterward, a corpus of words is created from scratch by inserting the preprocessed model elements iteratively. It is worth noting that a single element is not inserted if it is already included in the dictionary. In such a way, MORGAN encodes relevant information related to the application domain by embedding key features extracted from actual models. Furthermore, this component employs NetworkX,[13] a Python library that creates nodes and edges considering the structure of the parsed model. According to the format shown in Equation 6.10, each model is represented by a list of connected graphs in which each class is linked with corresponding elements.

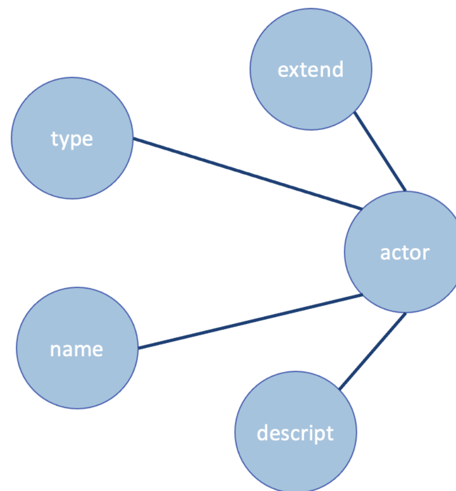


Fig. 6.14 Example of a stemmed graph.

An explanatory graph obtained by the *Encoder* component is depicted in Fig. 6.14. It is an encoding fragment of the metaclass *Actor* shown in Fig. 6.1b. In particular, the structure of the model is encoded by adding edges among the class *Actor* and the stemmed version of its attributes and references namely *Name*, *Description*, *Type*, and *Extends*, though such type of graph does not resemble the semantic relationships occurring in the actual model, i.e., an attribute can refer to a missing metaclass. Nonetheless, analyzing the structure of the model is enough to create the vocabulary that represents the knowledge base of the underpinning model. It is worth noting that the same format is kept to perform the final recommendations, thus delivering additional information about the type of the model element to the user.

6.2.2.3 Matrix Encoder

At this point, the *Matrix Encoder* component counts the occurrences of the items belonging to the obtained graphs. Each item has three required features, i.e., *adjacency_matrix*,

[12] https://www.nltk.org/_modules/nltk/stem/wordnet.html
[13] https://networkx.org/

node_labels, and *edge_labels*. The first feature is required to ensure that the graph is valid from the structural point of view. The second and third features are not mandatory in some certain contexts. For instance, e.g., *edge_labels* is not required in case the kernel algorithm does not employ them in the recommendation phase.[14] To feed the graph kernel, *Matrix Encoder* assigns a unique ID to each node of the graph to build a term-frequency matrix. In such a way, the most frequent items are considered at recommendation time. This component eventually calculates the sum of products between frequencies of common occurrences by employing the diagonal method to perform the dot product operation. It is worth mentioning that this strategy speeds up the diagonal calculation to reduce the computation time.

6.2.2.4 Vertex Histogram kernel

At this point, the underpinning engine can be fed with the computed graphs to retrieve the missing artifacts. To this end, MORGAN relies on the Grakel Python library that provides several graph kernel implementations [242]. We compute graph similarity by comparing the input model with all the elements in the training set. As discussed in Section 6.2.1, we replace the Weisfeiler-Lehman algorithm [239] employed in our previous work with the Vertex Histogram kernel strategy since the latter has been designed to handle the halting issue, even though the topological structure of the graph is not considered while performing the similarity measure. To mitigate this, we add the Matrix encoder component that produces an adjacency matrix given the produced graphs. In such a way, the structure of the graphs is preserved, thus enabling the computation of the kernel similarity.

In practice, given a pair of graphs G and G', let f, f' be the vertex histograms of the two graphs. The Vertex Histogram kernel is computed according to the following formula:

$$k(G,G') = \langle \mathbf{f}, \mathbf{f'} \rangle \tag{6.15}$$

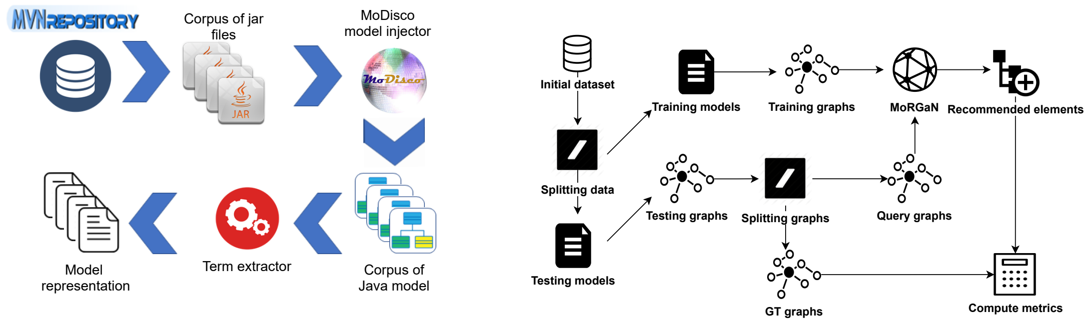
where $\mathbf{f}, \mathbf{f'}$ are the vertex label histograms obtained by applying the mapping function $\ell : \mathcal{V} \to \mathcal{L}$ that assigns labels to the vertices of the graphs.[15]

The rationale behind the selection of Vertex Histogram is as follows. The implemented version of the algorithm is linear in the number of vertices of the graph.[16] Moreover, through an empirical evaluation, we realized that it is much more timing efficient than the graph kernel adopted in the former version of MORGAN, i.e., Weisfeiler-Lehman strategies. Thus, Vertex Histogram has been chosen to compute the similarities among the nodes in graphs.

[14] In the scope of the presented work, we do not employ the *edge_labels* feature since homogeneous graphs are considered.

[15] This process is internally performed by the Python implementation offered by Grakel.

[16] https://ysig.github.io/GraKeL/0.1a8/kernels/vertex_histogram.html?highlight=vertex%20histogram



(a) Data gathering (b) Overall evaluation process.

Fig. 6.15 D_α dataset creation and overall evaluation process.

The employed kernel eventually retrieves an ordered list of modeling artifacts stored in the training set ranked by similarity scores. These values are computed by exploiting the Vertex histogram algorithm as described in Equation 6.15. In practice, given the context of the modeler encoded as a graph, MORGAN compares it with the graphs extracted from the training set. At the end of the process, a similarity score is assigned for each comparison and a ranked list of graphs is produced. The top-5 similar elements are extracted from this set to support two kinds of recommendation: *(i)* specification of missing structural features; and *(ii)* generation of (meta)classes that can be used to enhance the artifact under specification with further concepts. It is worth noting that the whole process is adopted for each type of model artifact, i.e., metamodels, Java models, and JSON schema. This means that there are three different training processes in which MORGAN learns a specific set of features needed to recommend the proper items.

Being built on these components, MORGAN provides recommendations including both metamodels, models, and JSON schema. The succeeding subsection introduces an explanatory example to show to which extent MORGAN is useful for a metamodeling task.

6.2.2.5 Explanatory example

Table 6.14 Retrieved items for the UMLDSL metamodel.

Context	Recommended item
Step	*attribute* finalState:EString *reference* **continuation:Step** *reference* initialState:initState *reference* finalStates:FinalState
Flow	*attribute* **finalizeFlow:EBoolean** *attribute* eventPatternId:EString *reference* initialState:initState *reference* finalStates:FinalState
Step	*metaclass* **StepAlternative:EClass** *metaclass* Automaton *metaclass* FSM *metaclass* mFSM

Table 6.14 shows the suggested structural features for the UMLDSL metamodel depicted in Fig. 6.1a. We consider two different metaclasses extracted from the artifacts under construction, i.e., *Flow* and *Step*. From the ranked list of structural features, we elicit the most relevant ones given the recommendation context, i.e., the bold items in Table 6.14. In particular, the reference *continuation* is the recommended item that the modeler can use to complete the partial metaclass *Step*. Similarly, the metaclass *Flow* could be enhanced with the *finalizeFlow* attribute. Concerning the recommendation of new classes, we consider again the class *Step* as testing. In this case, the retrieved item is the *StepAlternative* metaclass that can enrich the metamodel, even though it is not included in the complete one. We see that MORGAN produces items pertinent to the modeler's context.

Altogether, it is evident that the recommended items are helpful as they are relevant to the given artifact. In this respect, we conclude that for the explanatory example, MORGAN is able to provide the modeler with useful recommendations to complete the given metamodeling activities. In the next sections, we present the experimental methodologies as well as an empirical evaluation of the tool using real-world datasets to study its feasibility in practice.

6.2.3 Evaluation

We describe in detail the research objectives and the experimental configurations used to study MORGAN's performance. First, the research questions that we address in this paper are presented in Section 6.2.3.1. Afterward, in Section 6.2.3.2 we describe the datasets used in the evaluation. Finally, the validation methodology and evaluation metrics are detailed in Section 6.2.3.3 and Section 6.2.3.4, respectively.

6.2.3.1 Research questions

The following research questions are addressed to study the new version of MORGAN, comparing it with the previous one [68].

- **RQ$_1$**: *Does the preprocessing step contribute to a performance gain of MORGAN?* We investigate whether the introduced preprocessing augmentation improves the overall performances in terms of identified metrics, i.e., success rate, precision, recall, and F-measure.
- **RQ$_2$**: *How does the vertex histogram kernel function impact on the computational efficiency?* In an attempt to extend the MORGAN tool, we employed a tailored graph kernel based on the term-frequency matrix as the underlying recommendation engine.

This research question aims to validate if the proposed mechanism helps MORGAN reduce the time required to perform the recommendations.

- **RQ**$_3$: *How effective is MORGAN at recommending JSON schema elements?* To assess if the tool is able to support different application domains, we evaluate by considering a curated dataset composed of JSON schema models. Such type of data is widely used in MDE, and thus we suppose that the capability to work with JSON will allow MORGAN to gain popularity in real-world scenarios.
- **RQ**$_4$: *How is MORGAN's performance changed when working on ModelSet, a benchmarking dataset of metamodels and UML models?* To further study MORGAN's capabilities in recommending modeling artifacts, we considered two additional datasets extracted from ModelSet [149], a recently collected set of metamodels and models. In such a way, we can measure how the size of the data affects the tool's overall performance.

6.2.3.2 Datasets extraction

Recent studies in the domain provide large, curated collections of models and metamodels, leveraging to support ML-based tasks, e.g., classification of models and prediction of relevant modeling artifacts. In this extension, we consider five different datasets, which are named D_α, D_β, D_γ, D_δ, and D_ε. The first two datasets, i.e., D_α, and D_β were already used to evaluate the former version of MORGAN [68], and we re-used them to evaluate the fine-tuned version of MORGAN proposed in this extension. Furthermore, with the aim of showcasing the extensibility of our approach, we make use of three additional datasets, namely D_γ, D_δ, and D_ε. The first one is composed of JSON schema crawled from GitHub. Meanwhile, D_δ and D_ε have been extracted from ModelSet [149], a curated collection of models and metamodels. The selected datasets are explained in detail as follows.
✧ Concerning D_α, we extracted model representations of popular Java projects stored in the *Maven repository*[17] to build the D_α dataset. The whole process to obtain the required data is depicted in Fig. 6.15a. First, we selected the top eight popular categories, including *Apache*, *Build*, *Parser*, *SDK*, *Spring*, *SQL*, *Testing*, and *Web server*, among the most popular ones according to the Maven Tag Cloud.[18] Then, for each category, we crawled around the top 100 popular Java artifacts.

The whole process aims to create a balanced dataset composed of good-quality models. In such a way, we aim to build a curated collection of models that share common features as

[17] https://mvnrepository.com/
[18] https://mvnrepository.com/tags

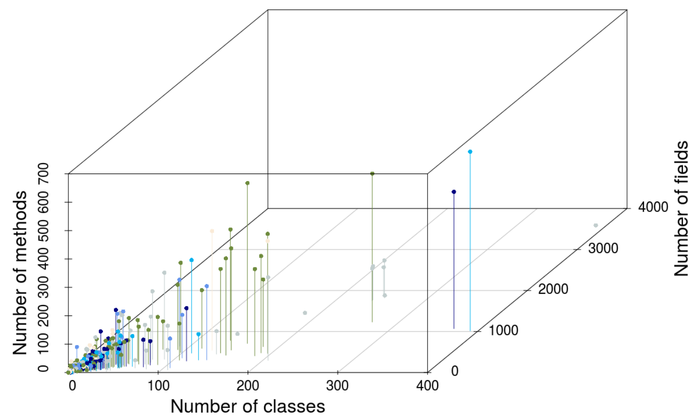


Fig. 6.16 D_α features.

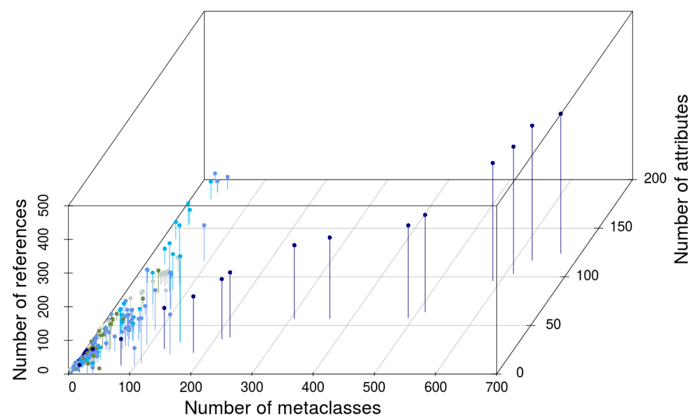


Fig. 6.17 D_β features.

much as possible. Having such similarities could improve the overall accuracy even though this cannot be granted at the beginning of the process. Figure 6.16 shows the statistics for the extracted models in D_α. In particular, the x, y, and z axes correspond to the number of classes, the number of methods, and the number of fields of the mined artifacts, respectively. Moreover, the colors of dots are used to represent the categories. It is evident that most of the models contain a small number of methods and classes, i.e., lower than 300 and 200, respectively. There is only one model with more than 1,000 fields, 390 classes, and 690 methods. The corpus of JAR files has been collected by employing Beautiful Soup,[19] a Python scraping library. Then, Java models have been generated from the collected corpus using MoDisco, an extensible framework that allows us to convert JAR files back to models.

[19] https://www.crummy.com/software/BeautifulSoup/bs4/doc/

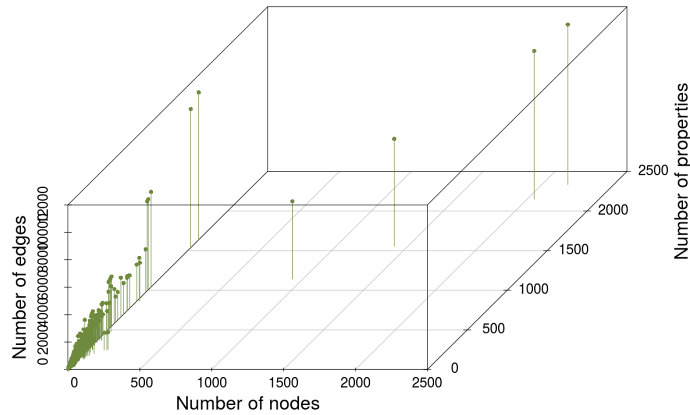


Fig. 6.18 D_γ features.

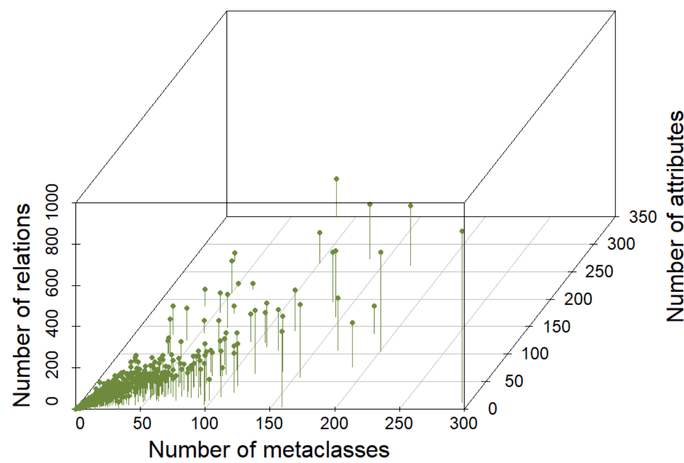


Fig. 6.19 D_δ features.

Since they are Java models, we extracted three different model elements from the MoDisco models, i.e., classes, methods, and fields. Finally, a model parser is used to represent the model as defined in Section 6.2.2. In the end, we collected a set of 581 unique model representations from the MVN Repository belonging to the top categories.

✧ To generate D_β, starting from an original set consisting of 555 labeled metamodels with nine different categories [300], we extracted metaclasses, attributes, and references from each *ecore* file using the Eclipse EMF utilities. Moreover, different quality filters have been also applied on the data, attempting to improve MORGAN's performance. In particular, we removed metaclasses having less than two elements, either attributes or references. Since

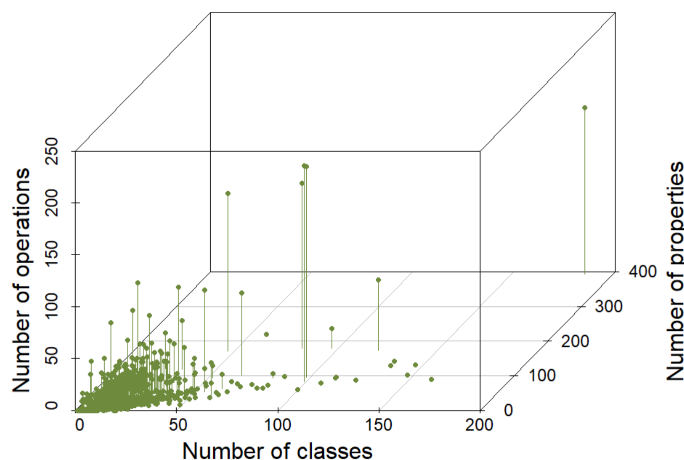


Fig. 6.20 D_ε features.

each metamodel is encoded as a set of graphs, having small ones may harm the overall performance. Thus, we eliminated metamodels belonging to the *Bug* category, which has only eight metamodels. Possible duplicate classes are also excluded to avoid any bias. The final dataset consists of the following eight categories: *Build*, *Conference*, *Office*, *PetriNets*, *Request*, *SQL*, *BibTex*, and *UML*. Figure 6.17 reports a summary related to the characteristics of D_β including the number of metaclasses (the x axis), the number of references (the y axis), and the number of attributes (the z axis). Like in Fig. 6.16, the category of metamodels is represented using colors. Though we do not directly employ the category to produce recommendations, it includes similar metamodels which help to represent the application domain.

✧ As discussed in Section 6.2.1, the definition of a JSON schema falls under the umbrella of modeling activity. In the light of such rationale, we introduced an additional dataset, i.e., D_γ, consisting of JSON schema mined from GitHub. Due to limitations imposed by the GitHub REST API [2], we mined a publicly-available GitHub archive dataset freely accessible in Google BigQuery [3]. First, we query metadata searching for files with *.json* extension across the repositories in [3], restricting the search to *.json* files whose pathname contains the string *schem*. The rationale behind it is twofold: *(i)* it limits the amount of data processed by the query on the GitHub archive dataset to avoid exceeding the free monthly per user quota (1TB) [3]; *(ii)* we aim to infer insights on *.json* files by its pathname. We then filtered out dangling entries, i.e., pairs where pathnames point to *.json* files that no longer exist, and existing *.json* files that do not contain the $*schema* keyword. Then, *duplicates removal*, *parsing*, and *conformance check* are performed.

The *duplicates removal* step computes a hash value for every mined file based on its content. The obtained hashes are used to build a hashmap where the key is the hash itself, and the value is the corresponding file. If a duplicated key occurs in the map, we assume that the corresponding *.json* file is a duplicate and we drop it. We distinguish between *parsing* and *conformance check* steps. The former checks if the JSON file is correctly serialized, while the latter validates the conformance of a schema considering its meta-schema definition.[20] Finally, we collected 2,872 distinct and valid JSON scheme and we summarize their features in Figure 6.18.

✧ ModelSet is a collection of 5,466 Ecore metamodels and 5,120 UML models manually labeled by the authors. This curated collection has been used in practice to enhance the performance of an existing search engine for models, namely MAR [148]. As we have shown in our previous work [69], the quality of the input data plays a key role in the performance of many model assistants. In particular, curated datasets with more similar metamodels allow recommender systems to improve their prediction performance, even if the size of such datasets is smaller than that of those randomly collected. In fact, ModelSet contains several low-quality models, i.e., 14% of metamodels and 13% of models are marked as *dummy*. Such artifacts are noisy, and they cause misprediction when being used as training data. More importantly, in ModelSet the models and metamodels are distributed among 50 and 80 categories, respectively. In other words, models and metamodels belonging to each category are not balanced, e.g., many categories include less than 3 models or metamodels. For instance, 61, 35, and 17 categories count 1, 2 and 3 metamodels, respectively; or 25, 9, and 3 categories count 1, 2, and 3 models, respectively. This means that the similarities between metamodels are not guaranteed in the dataset. Altogether, having such dissimilarities could negatively affect the overall recommendations. Therefore, we make use of the provided Python API to filter the initial list of artifacts according to a well-defined quality filter. Eventually, we extract D_δ and D_ε and describe their features in Figure 6.19 and Figure 6.20 respectively. For all considered datasets we get rid of small (meta)models and keep only the larger ones since MORGAN is a data-driven approach that heavily relies on the quality of training data.

For all considered datasets we get rid of small (meta)models and keep only the larger ones since MORGAN is a data-driven approach that heavily relies on the quality of training data.

[20] https://json-schema.org/specification.html

6.2.3.3 Settings

In the original study [68], we assessed the prediction performance of MORGAN by resembling the behaviors of a modeler[21] working at different stages of a modeling project *m*, by involving different configurations during the experiments [181]. To this end, we split *m* and use the rest as the modeler's context by varying two parameters, i.e., the number of considered classes and the number of the corresponding structural features. Starting from these definitions, we create four configurations to simulate different stages of modeling, i.e., from an initial specification to a mature one. We realized that MORGAN is capable of assisting the modeler in all considered scenarios, and the best accuracy scores were obtained when a mature context is considered [68]. In this work, we focus on assessing the contributions of the novelties introduced at the level of the recommendation engine, i.e., the additional preprocessing step and the vertex histogram kernel function. In particular, we are interested in measuring the contribution of each component separately and comparing the achieved results with the former version of MORGAN.[22] Table 6.15 summarizes the identified configuration by considering the aforementioned components.

Table 6.15 Configuration settings.

Configuration	Lemmatizer	Vertex Histogram
C_1	✗	✗
C_2	✓	✗
C_3	✗	✓
C_4	✓	✓

Configuration C_1 represents the MORGAN's original setting, equipped with a standard NLP pipeline and the WeisfeilerLehman kernel. Starting from this, we derive other configurations, i.e., C_2 and C_3 by introducing the lemmatization step and a novel kernel similarity respectively. Finally, by C_4 we combine all the newly introduced mechanisms to assess how the new tool advances compared to its preceding version by analyzing two different dimensions, i.e., the accuracy in terms of relevant results, and the delivery time. The former might benefit from the improvement of preprocessing phase as the data-driven tools strongly rely on input data to compute the outcomes. In contrast, the latter is affected by the complexity of the employed kernel function; thus, improving this component could speed up both the training and the query phases. In this respect, we expect that the *Lemmatizer* component increases the quality of the recommended items while the computation time might be reduced

[21]For the sake of presentation, the two terms "*modeler*" and "*developer*" are used interchangeably in the scope of this chapter.

[22]Replicating the experiments with all the configurations is out of the scope of the presented work. Thus, we selected the one that leads to a better accuracy, and used it to evaluate the novel aspect introduced in this extension.

by exploiting the *Vertex Histogram* kernel. To conduct the experiments, we split a dataset into two independent parts, namely a *training set* and a *testing set*. In practice, the training set represents the (meta)models that have been collected ex ante, they are available at developers' disposal. The testing set represents the metamodel being developed, or *the active project*. In this way, our evaluation simulates a real-world scenario: *the system is supposed to generate recommendations for the active metamodel based on the data mined from a set of existing metamodels*. In the evaluation, we adopted the k-fold cross-validation technique widely used in evaluating ML-based applications [210]. The overall process is described in Fig. 8.14 and it is applied on both the metamodel dataset and the model one presented in Section 6.2.3. Given the initial datasets, the splitting data operation is performed to obtain training and testing sets. In practice, the former represents the models that have been collected a priori to build the vocabulary (see Section 6.2.2), while the latter has been split into *GT graphs* and *query graphs*. In our previous work [68], we already evaluated the performance MORGAN with different *GT* and *query* sizes to mimic the behaviors of a modeler working at different stages of a modeling project. In the presented work, we resemble the situation where the metamodel under construction is almost complete, i.e., the modeler already defined the two third of classes and structural features. In particular, a single *query* graph represents the active context of the modeler who is defying classes and structural features for a specific model. Meanwhile, the *GT* graph is the elicited part extracted from the original model that should be defined by the modeler to complete the partial model. Even though this splitting strategy could lead to possible inconsistencies, we carefully encode the original models to avoid any broken references.

6.2.3.4 Metrics

6.2.4 Experimental results

To investigate the potential contributions of the conceived extension, we replicate the study conducted in the original MORGAN paper [68] by analyzing the performances obtained with the abovementioned enhancements, i.e., lemmatization preprocessing and the Vertex Kernel strategy. The evaluation adheres to the configuration settings discussed in Section 6.2.3 to avoid any bias in the comparison. We investigate if the mechanisms presented in Section 6.2.2 contribute to an increase in the overall accuracy in terms of the selected metrics, i.e., success rate, precision, recall, and F-measure, as well as in a gain considering the delivery time. The experimental results are reported using violin boxplots, representing both boxplot and density traces. This aims to bring a more informative indication of the distribution's shape [109], enabling us to comprehend better the magnitude of the density. To this end,

we report and analyze the experimental results by answering the three research questions introduced in Section 6.2.3.

RQ_1: *Does the preprocessing step contribute to a performance gain of MORGAN?* To assess the contribution of the lemmatizer component, we compare the former version of the tool with MORGAN equipped with the new NLP module, namely configurations C_1 and C_2 respectively, considering the two recommendation tasks related to the modeling activity, i.e., providing model classes and class members.

▷ **Recommending model classes.** The comparison between the two identified configurations of MORGAN employed to support model classes recommendation are depicted in Fig. 6.21. It is evident that by Configuration C_2, MORGAN increases the overall accuracy even though the measured improvement is negligible in terms of the considered metrics, i.e., the distribution of the relevant items are similar to the preceding version of MORGAN. For instance, the violin boxplots representing the success rate values shown in Fig. 6.21a span from 0.10 to 0.60 for both the considered configurations. Despite this, by C_2, MORGAN slightly improves the relevance of the delivered items since they are more uniformly distributed than the ones retrieved by C_1. This finding is confirmed by analyzing the values obtained by the other metrics, i.e., precision, recall, and F-measure. While MORGAN suffers from some degradation of performance in terms of precision with C_2, we measure higher values for recall, meaning that the new version of the tool reduces the number of false negatives. On the one hand, the violin boxplot for C_1 shown in Fig. 6.21b spans from 0.0 to 0.20 while the one representing results for C_2 reaches 0.10 as a maximum value. On the other hand, Fig. 6.21c shows that MORGAN obtains a better recall by C_2 compared to C_1, i.e., the variance of the results is reduced by adopting the lemmatizer component. The F-measure values depicted in Fig. 6.21d confirm that the recommendation of model classes benefits from the introduced enhancements, i.e., recall mitigates the lower results obtained by the precision.

▷ **Recommending class members.** The improvement is more evident by analyzing the results obtained in the second modeling task, i.e., the recommendation of class members, as depicted in Fig. 6.22. Take as an example, the obtained success rate depicted in Fig. 6.22a ranges from 0.30 to 0.80 and from 0.40 to 1.00 by using C_1 and C_2, respectively. This demonstrates that the lemmatizer component is capable of increasing the relevance of the returned items as the success rate is increased by 10% on average. Such an improvement is confirmed by analyzing the results obtained for precision, recall, and F-measure showed in Fig. 6.22b, Fig. 6.22c, and Fig. 6.22d, respectively.

More specifically, the precision reaches the maximum value of 0.60 when C_2 is considered. Meanwhile, the former version of MORGAN represented by C_1 earns a maximum precision

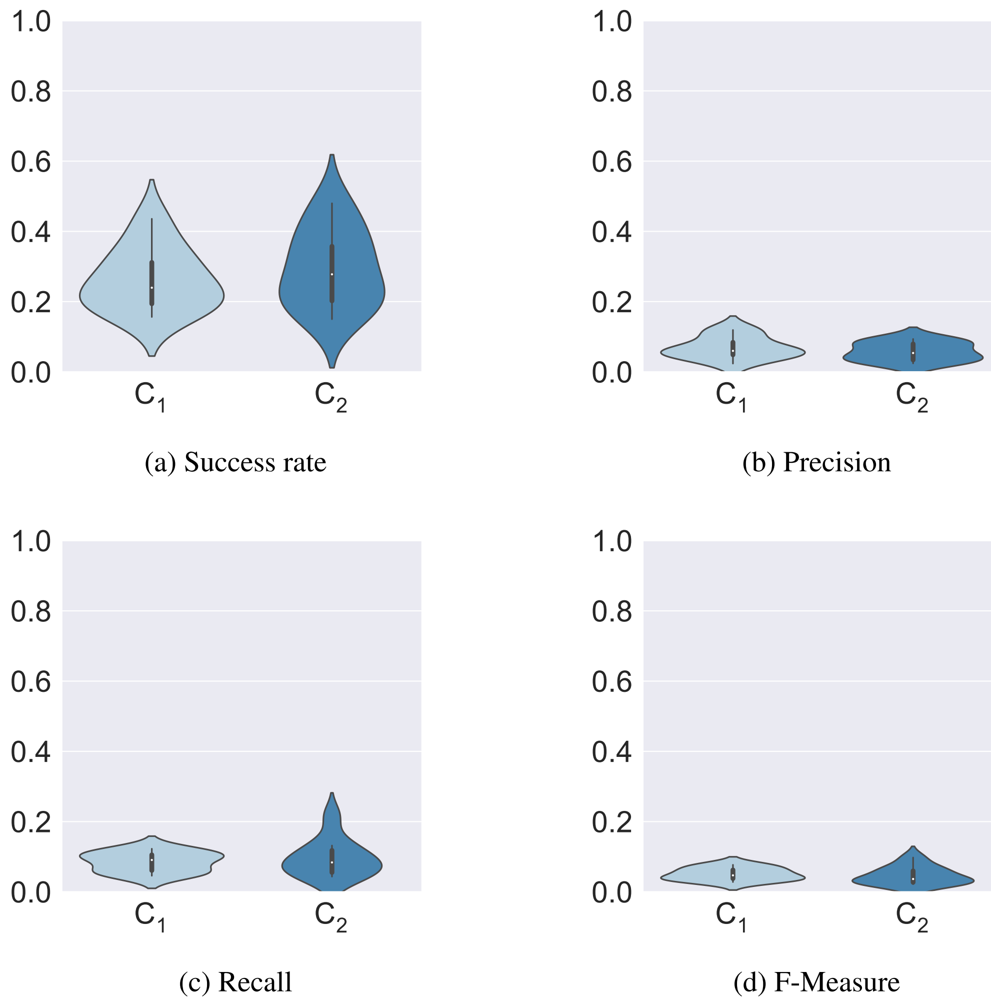


Fig. 6.21 Evaluation scores for recommending model classes.

of 0.50 even though the distribution of the items is similar in terms of variance. Similarly, the new NLP component is capable of increasing the recall by 10% on average with respect to the former version of the tool represented by C_1's violin boxplot. Overall, adopting C_2 facilitates a better performance although the delta is minimal. This issue has been already threatened in the original work by analyzing the similarity among the models belonging to D_α.[23] We realized that the similarity of the considered models is very low, thus undermining the capability of the approach to be more effective. This is quite expected as MORGAN is a data-driven tool that strongly relies on the quality of the input data. Therefore, introducing an

[23] In the scope of the presented work, we do not report the similarity values among the considered models. The interested readers are kindly referred to the previous version of MORGAN [68] for more detail

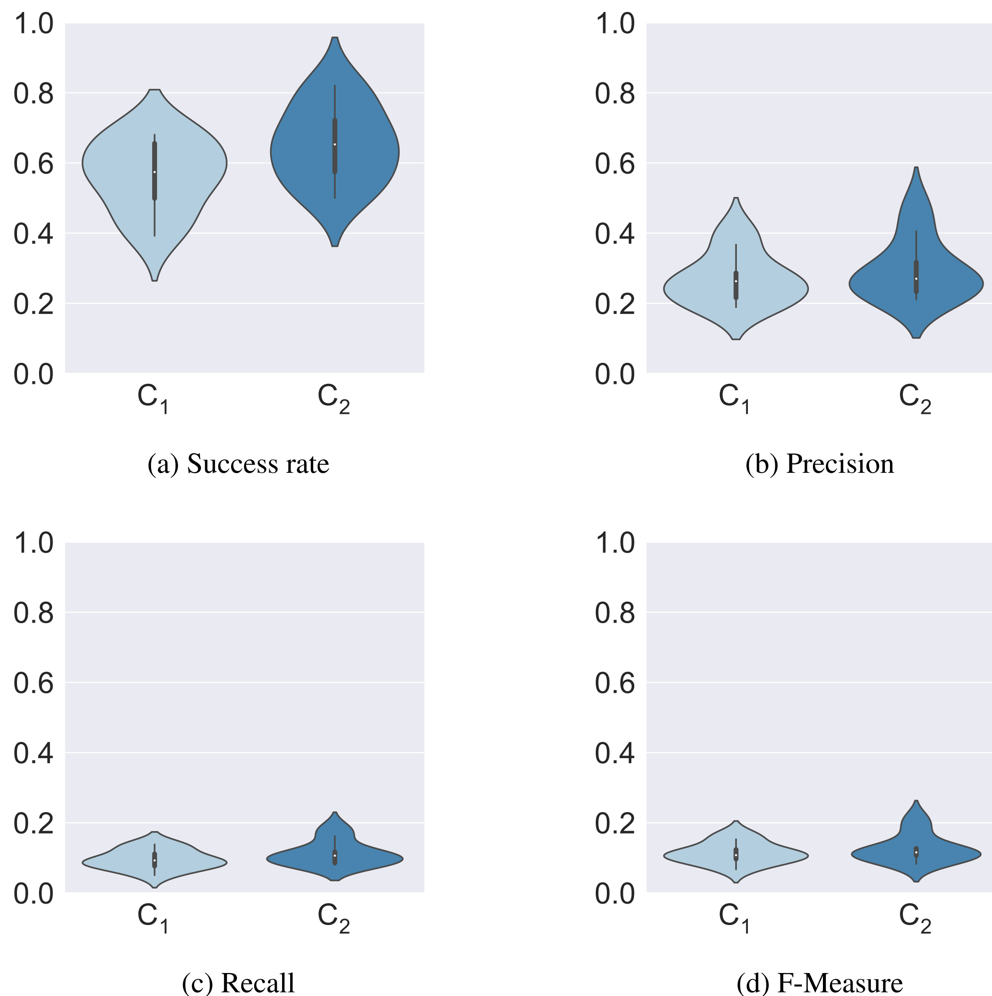
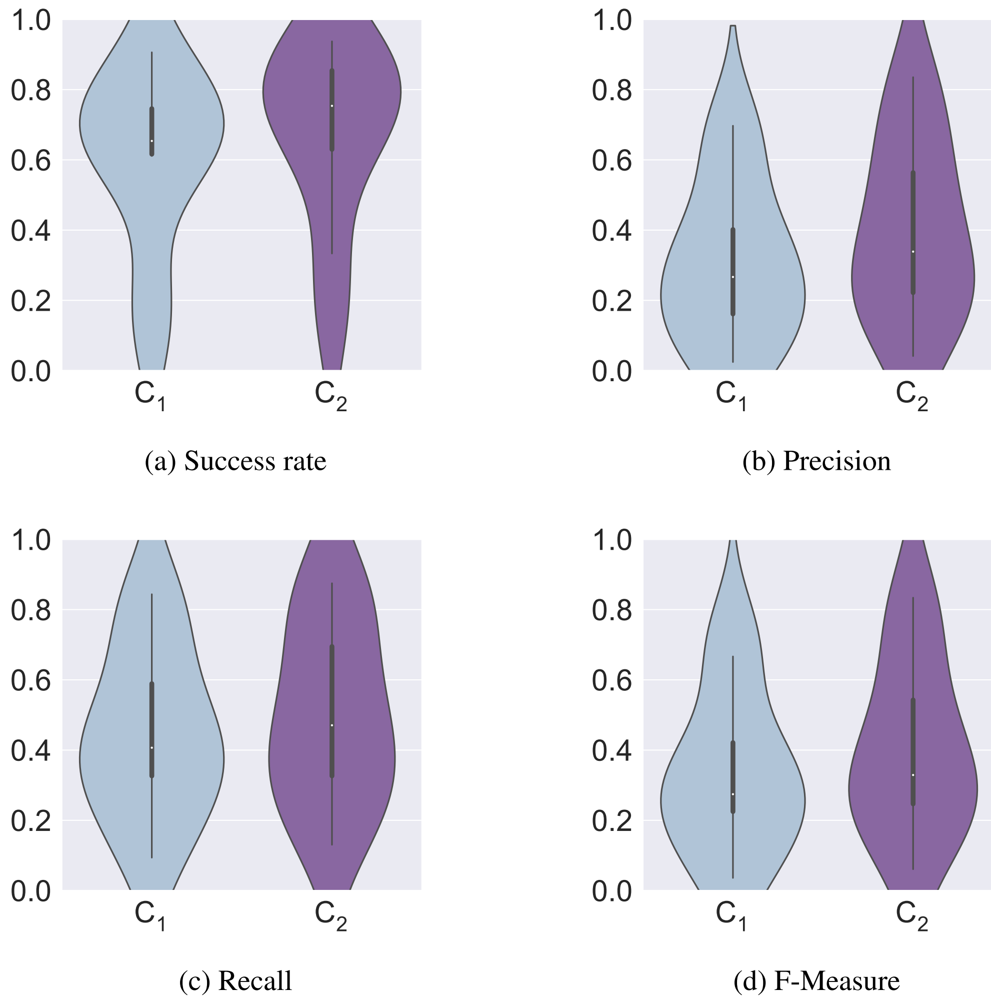


Fig. 6.22 Evaluation scores for recommending class members.

additional preprocessing step contributes to lead better performance even with not remarkable results.

▷ **Recommending metaclasses.** Similar to the previous analysis, Fig. 6.23 shows the comparison of the old MORGAN, i.e., C_1 and the new one, i.e., C_2 in recommending metaclasses. It is clear that the proposed approach performs better in both recommendation contexts, i.e., all the metrics are improved by 10% on average. By considering the recommendation of metaclasses, the boxplots show that MORGAN delivers more relevant items compared to the former version, i.e., the overall variance is reduced by adopting the enhanced approach. Such an improvement becomes evident by analyzing the success rate metric shown in Fig. 6.23a. While using the original approach the corresponding violin plot spans from 0.20 to 0.80, the recommended metaclasses are concentrated in the upper part of the diagram by adopting

the extended version of MORGAN. This means that the variance of the elements is reduced since they are distributed from 0.30 to 0.90.



(a) Success rate

(b) Precision

(c) Recall

(d) F-Measure

Fig. 6.23 Evaluation scores for recommending metaclasses.

Similarly, the precision, recall, and F1-measure values computed with C_2 are more equally distributed with respect to the results obtained in the previous work, represented by C_1. For instance, the distribution of the precision values presented in Fig. 6.23b is centered around 0.20 using C_1, implying that most of the metaclasses are not suitable for a given context. In contrast, adopting C_2 improves the quality of the retrieved artifacts, i.e., the corresponding violin span homogeneously. The same trend can be observed for the other metrics, i.e., recall and F-measure shown in Fig. 6.23c and Fig. 6.23d. Altogether, this means the new preprocessing component contributes to increasing the overall quality of the retrieved artifacts.

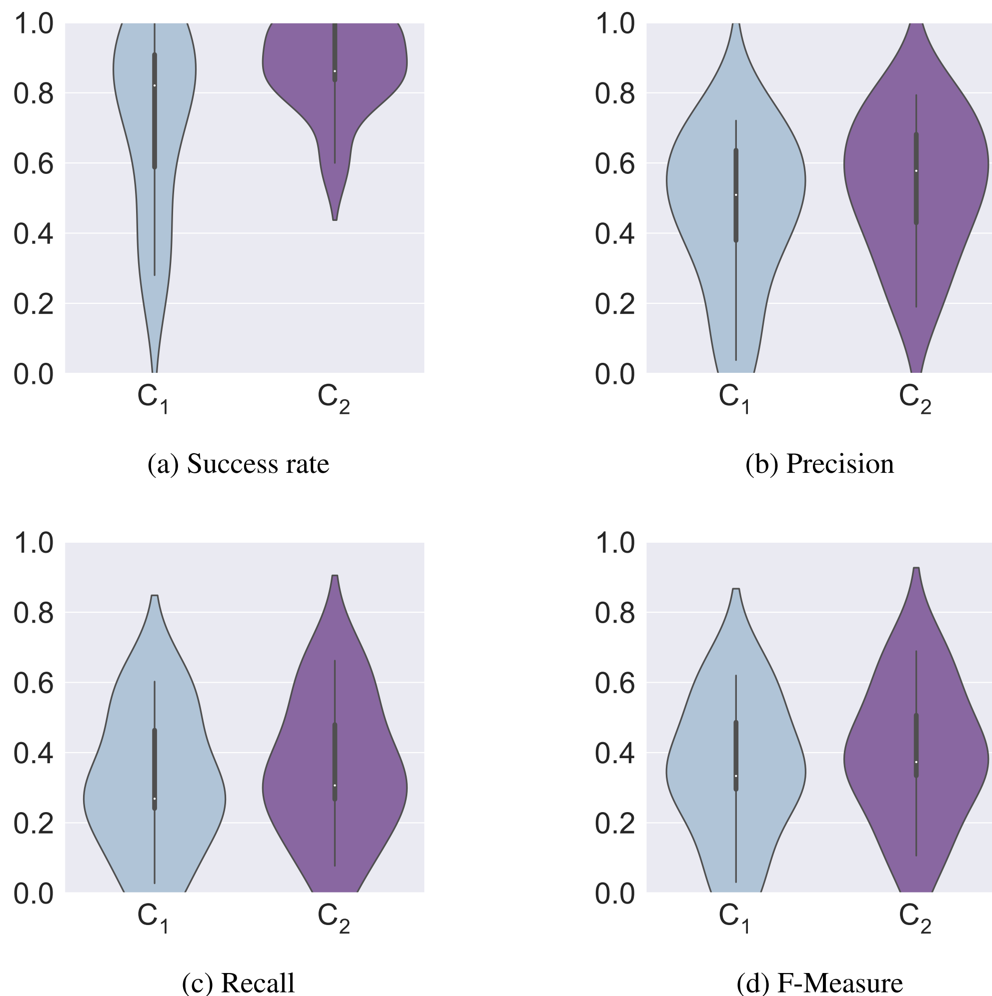


Fig. 6.24 Evaluation scores for structural features.

▷ **Recommending structural features.** Similar to D_α, the contribution of the advanced preprocessing step is more evident when structural features are considered. In particular, the success rate values obtained by MORGAN equipped with the lemmatizer component strongly confirm our findings, since the scores are concentrated on the [0.60; 1.00] range. Meanwhile, adopting C_1, i.e., the former version of MORGAN, produces less relevant artifacts as the corresponding boxplot is scattered from 0.0 to 1.00 as shown in Fig. 6.24a. Roughly speaking, this means that the extended version of MORGAN comes in handy for a larger set of testing contexts, improving the overall quality of suggested structural features. The other considered metrics remark this improvement, i.e., the distribution of values obtained with MORGAN are greater than the ones of the original work, though the delta is less noticeable compared to the success rate metric. As shown in Fig. 6.24b, the

precision values obtained by using C_2 are very similar to those computed by using C_1, i.e., the former version of MORGAN. Nonetheless, we observe that most of the values are centered around 0.50 and 0.60 considering C_1 and C_2 respectively, meaning that the lemmatizer component improves the quality of the retrieved items even in a small percentage of cases. This finding is confirmed by the recall values as the violin plot depicted in Fig. 6.24c have almost a similar shape. Figure 6.24d eventually summarizes the obtained enhancements by showing the F-measure results for the two examined configurations. It is worth noting that C_2's violin plot reaches 0.85 as a maximum while C_1 distribution stops at 0.80. This can be explained by observing the recall boxplot since the one representing precision has the same shape. Thus, the decrease of the variance in the recommended items distribution is almost led by recall, as the F-measure score represents the harmonic mean of the two aforementioned metrics.

To better understand the contribution of the new configuration, we compute two widely used statistical tests on the two populations, i.e., Wilcoxon rank sum test adjusted p-values and Cliff's d. Table 6.16 summarizes the obtained results considering the average values of the metrics for both D_α and D_β datasets. Concerning the former, we observe that the introduction of the new kernel has a negligible effect on the results as confirmed by the two statistical tests. Furthermore, the aggregated precision, recall, and F1 values are almost the same in both cases, meaning that adopting Configuration C_2 does not lead to a better performance. Such a negative result can be explained by the heterogeneity of the Java models belonging to D_α. Essentially, the introduction of the new kernel is not enough to obtain better results when D_α is considered. In contrast, we observe an improvement when MORGAN is equipped with the Vertex Histogram kernel on D_β, i.e., MORGAN achieves better average success rate, precision, and recall scores than the ones achieved from the previous version for both metaclasses and structural features recommendations. Though the improvement is not statistically significant, the aggregate values show that MORGAN equipped with C_2 leads to better results by considering a more curated dataset.

> **Answer to RQ$_1$.** Compared to the former version, MORGAN improves its performance by adopting refined preprocessing strategies, i.e., lemmatization instead of stemming. Though such improvements impact mostly on success rate, the enhanced preprocessing increases the probability of retrieving valuable artifacts.

RQ$_2$: *How does the vertex histogram kernel function impact on the computational efficiency?* With the aim of assessing to what extent the time of recommendations can be improved, we compared the former version of the system with the one that exploits the Vertex Histogram kernel as a recommendation engine, i.e., considering C_1 and C_3. To this end,

Table 6.16 Average prediction scores, Wilcoxon rank sum test adjusted p-values and Cliff's d results.

	D_α				D_β			
	Class members		**Classes**		**Structural features**		**Metaclasses**	
	C_1	C_2	C_1	C_2	C_1	C_2	C_1	C_2
Avg. Success rate	0.36±0.48	0.36±0.47	0.22±0.41	0.21±0.41	0.97±0.15	0.99±0.09	0.66±0.47	0.73±0.44
Avg. Precision	0.05±0.13	0.06±0.13	0.04±0.11	0.04±0.09	0.48±0.30	0.70±0.28	0.50±0.43	0.56±0.41
Avg. Recall	0.03±0.12	0.03±0.08	0.08±0.20	0.07±0.20	0.40±0.26	0.74±0.25	0.41±0.38	0.51±0.38
Avg. F-Measure	0.03±0.10	0.03±0.07	0.04±0.10	0.04±0.09	0.57±0.23	0.86±0.22	0.39±0.40	0.54±0.42
Wilcoxon p-value[2]	0.6486		0.2149		0.0856		0.6812	
Cliff's d results [2]	-0.01295 (n)[1]		-0.0122 (n)[1]		-0.0142 (n)[1]		-0.0877 (n)[1]	

[1] magnitude values l:large, s:small, n:negligible.
[2] Both Wilcoxon rank sum test adjusted p-values and Cliff's d results are computed on the success rate scores.

we analyze the computational efficiency in terms of *(i)* training time needed to learn the encoded features in the models and *(ii)* testing time, namely the time needed to get a set of recommendations given the modeler's context. Similar to the previous research question, we conducted such an evaluation by considering the two datasets of the original work, i.e., D_α and D_β. Furthermore, we investigate how the type of recommended item could affect the system from the computational point of view, i.e., if recommending metamodel classes or their structural features can lead to a different execution time. Table 6.17 summarizes the results of the comparison between the two aforementioned configurations when models are considered.

The measured time shows that augmenting MORGAN with Vertex Histogram results in better computational efficiency. Considering the training phase, the benefit of adopting the novel kernel is more evident for the recommendation of class members, i.e., the required time decreases from $7,815$ to 923 seconds on average. Furthermore, this operation takes $7,570$ and $1,242$ seconds for model classes when C_1 and C_3 are enabled respectively. The Vertex Histogram contributes also to reducing the whole testing time, i.e., Configuration C_3 takes 101 and 166 seconds for classes and their members respectively. Meanwhile, the former version of MORGAN that adopts C_1 requires 689 seconds to recommend model classes and 868 seconds for the considered members. Furthermore, we measure the time required to produce the recommendations for a single model. It is evident that Vertex Histogram performs better compared to the Weisfeiler-Lehman kernel, i.e., the needed time decreases from 4 to 0.6 and from 5 to 1 seconds for classes and members recommendations respectively.

Concerning D_β, the results confirm that equipping MORGAN with Vertex Histogram helps speed up the overall recommendation process, i.e., MORGAN gets a better prediction when running with C_3 instead of C_1 by both considered metrics. In particular, the time required using C_1 for the training phase is 120 seconds on average for the metaclasses while adopting C_3 needs 17 seconds considering the same amount of data. Similarly, the same

Table 6.17 Timing performance on D_α (seconds).

	Classes		**Class members**	
Operation	C_1	C_3	C_1	C_3
Preprocessing	7,570	**1,242**	7,815	**923**
Testing	689	**101**	868	**166**
Single rec.	4.0	**0.6**	5.0	**1.0**

trend can also be seen for structural features since MORGAN equipped with the new kernel module reduces the whole training time from 153 to 51 seconds.

Table 6.18 Timing performance on D_β (seconds).

	Metaclasses		**Structural features**	
Operation	C_1	C_3	C_1	C_3
Preprocessing	120	**17**	153	**51**
Testing	14	**9**	14	**9**
Single rec.	1.0	**0.2**	1.5	**0.3**

By the testing operation, the tool requires 14 and 9 seconds when C_1 and C_3 are adopted, respectively. It is worth mentioning that the computed time is the same for both metaclasses and structural features. This finding can be explained by considering the average dimension of the metamodels belonging to the D_β dataset, i.e., they include a small number of structural features of each metaclass. Therefore, the recommendation phase takes almost the same time for the two aforementioned metamodel artifacts.

This claim is confirmed by analyzing the time needed for a single recommendation, i.e., it is almost the same for a single model considering both kernels. It is evident that C_3 leads to better performances compared to the results obtained by running MORGAN with C_1. In particular, recommending a set of classes and structural features requires 0.2 and 0.3 seconds respectively by adopting C_3. Meanwhile, MORGAN equipped with the Weisfeiler-Lehman kernel takes 1.0 and 1.5 seconds on average for the two recommendation tasks.

Altogether, Vertex Histogram improves the computation efficiency in all the considered scenarios, i.e., the recommendation of models and metamodels artifacts. However, MORGAN's overall accuracy may decrease as we are employing a different technique. Therefore, we replicate the analysis conducted in the previous research question by comparing C_1 and C_3 in terms of the considered metrics, i.e., success rate, precision, recall, and F-measure.

Table 6.19 shows the comparison by considering the average values of the mentioned metrics considering models, i.e., D_α.

Table 6.19 Comparison between C_1 and C_3 considering D_α.

	Classes		**Class members**	
Metrics	**C_1**	**C_3**	**C_1**	**C_3**
Success rate	0.21	**0.23**	0.63	**0.64**
Precision	0.03	**0.05**	0.27	**0.29**
Recall	0.08	**0.09**	0.10	**0.11**
F-measure	0.04	**0.06**	0.11	**0.12**

It is evident that the novel graph kernel preserves the overall accuracy of the original work, i.e., all the metrics are improved on average. In particular, the introduction of Vertex Histogram leads to better results in recommending the two types of model artifacts, i.e., classes and their corresponding members. For instance, the success rate measured for model classes passes from 0.21 to 0.23 when C_3 is considered. Similarly, the other metrics are improved by 1% on average with respect to C_1. Similarly, MORGAN equipped with the new kernel strategy achieves better performance when class members are considered even though the delta is negligible. Nonetheless, Vertex Histogram aims to improve computational efficiency in the first place since better performance in terms of observed metrics is obtained by means of the new preprocessing component.

Table 6.20 Comparison between C_1 and C_3 considering D_β.

	Metaclasses		**Structural features**	
Metrics	**C_1**	**C_3**	**C_1**	**C_3**
Success rate	0.60	**0.62**	0.72	**0.78**
Precision	0.30	**0.33**	0.46	**0.49**
Recall	0.44	**0.45**	0.31	**0.32**
F-measure	0.33	**0.35**	0.34	**0.36**

Table 6.20 confirms that MORGAN's prediction scores are not hampered by the introduction of the new graph kernel. The table shows that the examined metrics are improved up to 2% apart from the success rate measured for structural features, i.e., its score reaches 0.78 using C_3 while using C_1 yields 0.72 as the maximum. Such an improvement can be explained by considering the strong similarity among the considered metamodels. Altogether,

the observed results demonstrate that the contribution of the Vertex Histogram nurtures better results with respect to accuracy alongside the time required for the whole recommendation process.

Answer to RQ$_2$. Equipping MORGAN with the Vertex Histogram kernel helps improve the computation efficiency. Moreover, the overall prediction accuracy has also been slightly enhanced, meaning that the kernel strategy contributes to a performance gain, albeit marginal.

RQ$_3$: *How effective is MORGAN at recommending JSON schema elements?* To examine the generalizability of the tool, we assess MORGAN's capability of supporting the two modeling completion tasks over the D_γ dataset composed of JSON schema, namely root properties and the nested ones that can be mapped to metaclasses and structural features respectively as discussed in Section 6.2.1. Thus, we conduct the same evaluation presented in the previous subsections by using the configuration that embodies the novel components presented in this extension, i.e., Configuration C$_4$, that includes the lemmatizer component and the Vertex Histogram kernel.

▷ **Recommending JSON root properties.**
Figure 6.25 shows the results obtained when MORGAN is employed to recommend JSON schema properties given an incomplete model. It is worth mentioning that the success rate score span from 0.2 to 1.00. This means that MORGAN recommends at least one correct property in almost all of the examined contexts. In contrast, we experience some performance degradation when the other metrics are considered. For instance, the precision boxplot ranges from 0.0 to 0.40, i.e., relevant properties are recommended with a low probability on average. In this respect, these results are similar to the ones obtained for the models belonging to D_α. This finding is confirmed by the recall values as the number of items properly delivered is higher, i.e., the maximum value reaches 0.60. The distribution of the F-measure values resembles the precision ones, suggesting that false positives have a negative effect on the overall performances. By carefully inspecting the results, we observe that the D_γ dataset is very heterogeneous since all the schema have been extracted from GitHub repositories. Thus, such degradation of performance is due to the nature of the considered data even though the novelties introduced in the approach mitigates this issue.

▷ **Recommending JSON nested properties.** Figure 6.26 summarizes the results obtained by MORGAN in recommending nested JSON properties. Similar to the two examined datasets in Section 6.1.4, the tool obtains better results compared to the root properties recommendations. For instance, the success rate achieves 0.70 on average as we can observe from the corresponding boxplot. Furthermore, the majority of the values span from 0.60 to 0.90, meaning that nested properties have been recommended in most of the cases. A similar

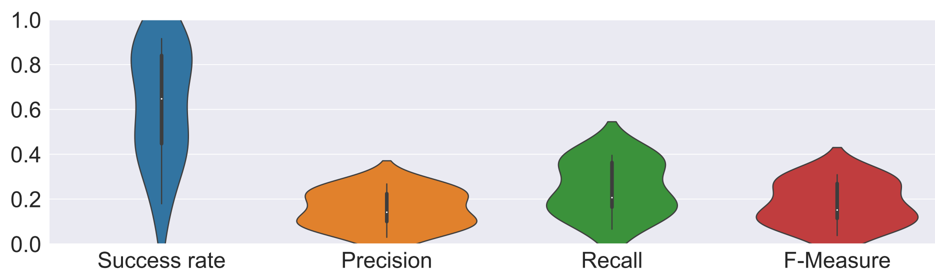


Fig. 6.25 Evaluation scores for recommending JSON root properties.

trend can be observed by analyzing the precision violin plot even though the average value is around 0.60. However, MORGAN suffers from some degradation in performance in terms of recall, i.e., the corresponding violin plot spans from 0 to 0.80, with an average value of 0.30. In other words, the system fails to detect false negatives when JSON nested properties are considered. This impacts negatively on the F-measure metric as the corresponding violin plot has a similar shape compared to the recall one. Despite this, the conducted study aims to demonstrate the capability of MORGAN in recommending different modeling artifacts other than Ecore models and class diagrams. It is our firm belief that improving the quality of the considered JSON schema will lead to a better accuracy. For instance, we can enhance the feature extraction process to include more relevant data embedded in a JSON schema, e.g., the type of the properties.

At the current stage of development, the system can provide *(i)* attributes and relationships to enrich a class or *(ii)* a list of similar classes considering the corresponding structural features. In principle, we could adapt the conceived parser module to extract relevant features from any kind of modeling artifacts. Therefore, MORGAN can possibly support the completion of a state machine model under construction if being properly trained with models, i.e., a dataset composed of state machine models with a decent number of transitions. However, the final accuracy depends a lot on the quality of training data. Altogether, completion of state machine models is possible under certain conditions, i.e., the availability of a proper dataset, and the refactoring of the parser component. This, however, needs to be validated with real-world datasets, and we consider it as our future work.

Answer to RQ$_3$. MORGAN succeeds in supporting JSON schema completion even though we experienced some degradation in performance for some metrics, i.e., the recall scores are very low. Such negative results might be mitigated by improving the quality of the input schemas.

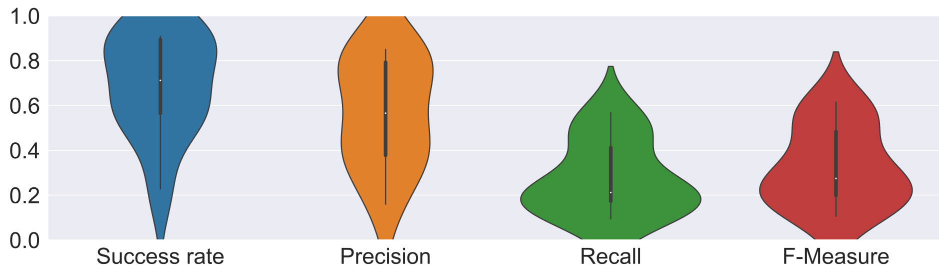


Fig. 6.26 Evaluation scores for recommending JSON nested properties.

To further study the tool's performance on a recent data, we extract two different datasets from ModelSet, namely D_δ and D_ε, by using the provided API. Table 6.21 summarize the results obtained by MORGAN on D_δ. Similar to RQ_3, we employ Configuration C4 to conduct the experiment for this research question. It is worth noting that the results are similar to the ones obtained FOR D_α, meaning that the degree of similarity among the artifacts is almost the same. Concerning D_δ, MORGAN obtains better performances in recommending metaclasses, i.e., all the scores are higher compared to the ones obtained for the structural features. In particular, the recall achieves 0.45 in recommending metaclasses while it stops at 0.28 for structural features. In other words, the number of false negatives is fewer when the metaclasses are considered. Concerning the time for a single recommendation, MORGAN is very fast, i.e., retrieving the suggested item requires only 0.04 and 0.03 seconds for metaclasses and structural features respectively. Nevertheless, the required time strongly depends on the size of the considered artifacts, thus leading to some scalability issues if a larger dataset is considered. In fact, MORGAN is faster on D_δ compared to D_α because the considered metamodels are smaller in terms of the number of attributes and relationships.

Table 6.21 Evaluation scores for D_δ.

Metric	**Metaclasses**	**Structural features**
Success rate	0.62	0.59
Precision	0.34	0.15
Recall	0.45	0.28
F-Measure	0.31	0.15
Single rec.	0.04	0.03

A similar trend can be observed for D_ε, i.e., recommending UML classes leads to better performances as shown in Table 6.22. Even though the success rate is slightly lower, i.e., 0.58 and 0.62 for classes and class members respectively, the other metrics confirm that MORGAN obtains better results in recommending the class entities compared to the former model dataset, i.e., D_α. This is quite expected since we extracted those models

by using MoDisco from Java projects while D_ε is composed of a curated list of UML models. Despite this, the scalability issue is confirmed by observing the required time for a single recommendation, i.e., MORGAN takes 0.20 seconds to suggest a list of relevant UML classes. Though the time required for the class attributes is slightly lower, i.e., 0.13 seconds, it is evident that augmenting the complexity of the graphs can hamper the timing performance of the tool. Altogether, the conducted analysis suggests that the dataset curation process contributes to achieving better results even though the scalability of the approach is compromised.

Table 6.22 Evaluation scores for D_ε.

Metric	Classes	Class members
Success rate	0.56	0.61
Precision	0.24	0.15
Recall	0.33	0.19
F-Measure	0.20	0.13
Single rec.	0.21	0.13

Answer to RQ$_4$. The obtained results obtained by considering ModelSet are comparable with the ones reported in the previous analysis, though MORGAN suffers from the scalability issue if complex graphs are considered.

6.2.5 Threats to validity

We discuss the threats that could impact on the validity of the study's outcomes. Moreover, we also identify possible countermeasures to mitigate them.

Threats that arose in the former work *internal validity* were related to two aspects, i.e., the graph kernel similarity and the employed encoding scheme, including the preprocessing of the input models. Concerning the former, we enhance the underpinning kernel similarity by adopting the Vertex Histogram technique. In such a way, we increase the computational efficiency by reducing the whole recommendation time even for large graphs. Concerning the latter, the preprocessing phase might miss relevant data, i.e., stemming does not consider the semantics of the examined terms, leading to possible loss of features during the encoding phase. To minimize the threat, we employed lemmatization in the preprocessing pipeline to augment the overall accuracy of the tool. Moreover, the conducted evaluation on the ModelSet datasets reveals that MORGAN suffers from scalability issues if the size of the training data is increased. To mitigate this threat, further study on the underpinning algorithm is needed, and we leave this as a possible future work. The selection of JSON schema as modeling activity may hamper the *external validity* of our findings. Though they conform

to a well-defined metamodel, the internal structure is completely different from that of the artifacts examined in the original work. To tackle this issue, we adapted MORGAN's encoder component by introducing a tailored parser for JSON schema to obtain the same format used for the other artifacts, i.e., models and metamodels. Furthermore, the results in terms of accuracy might be undermined by the quality of the considered schemas, i.e., the similarity among the JSON belonging to the dataset. We mitigate this threat by applying a set of quality filters on the JSON schemas crawled from GitHub i.e., duplicates removal, parsing, and conformance check

6.3 Conclusion

Modelers are facing an overload of information in their daily tasks, and this triggers the need for decent machinery to assist them in choosing suitable sources of information. Though various modeling frameworks are in place, there is still a lack of automated assistance which can help modelers ease the burden of the modeling activities.

In this chapter, we present two recommender systems that exploits two different automatic techniques to assist modelers, i.e., collaborative filtering and graph kernels. MemoRec covers the specification of metamodels by encoding their contents in four different schemes. Afterward, the tool built rating matrices and applied a syntactic-based similarity function to predict missing items, i.e., classes and structural features. An evaluation on two independent datasets, i.e., $\mathbf{D}_1$ and $\mathbf{D}_2$, and four encoding schemes, i.e., SE_s, IE_s, SE_c, and IE_c, exploiting ten-fold cross-validation demonstrates that MemoRec is able to provide decent recommendations.

We made a step further by proposing MORGAN, a modeler assistant that supports the specification of both metamodels and models by relying on graph kernels. The first version of the tool employs the Weisfeiler-Lehman kernel to support the completion of metamodels and class diagram extracted from Java projects by using Modisco tool. In this dissertation,we present an extended version that introduces the lemmatization step in the preprocessing phase. Furthermore, we equipped the recommendation engine with a more efficient kernel similarity function, helping the system to obtain more relevant results in less time. An empirical evaluation of five real-world datasets demonstrated that MORGAN is applicable in different application domains, even though we experiment the scalability issue when a larger training set is considered.

Chapter 7

Challenges and lessons learned from the conceived RSSEs

So far, we have presented a series of RSSEs that support different applications domain. Besides the ones that are part of CROSSMINER architecture, a set of additional software assistants have been developed and evaluated following the methodology framework presented in Chapter 3. Although the presented approaches obtained good performances, we experienced various issues during the actual development phase of such new additional systems. For instance, the lack of proper training data to support modeling activity is an open challenge that we didn't address in the CROSSMINER project. Therefore, this chapter discusses the challenges and lessons learned elicited from the developed RSSEs. We present the takeaways of the CROSSMINER by reporting part of the content published in [67], including the main design features elicited from the existing literature. In particular, development challenges (DC) have been grouped

In the following sections, such steps are described in detail. For each of them, we discuss the challenges we had to overcome and the difficulties we had while conceiving the tools as asked by the projects' use-case partners. The methods we employed to address such challenges are presented together with the corresponding lessons learned.

An overview of all the challenges and lessons learned are shown in the map depicted in Fig. 7.1. For the sake of readability, challenges related to the requirement, development, and evaluation phases are identified with the three strings *RC*, *DC*, and *EC*, respectively, followed by a cardinal number. Similarly, the lessons learned are organized by distinguishing them with respect to the requirement (*RLL*), development (*DLL*), and evaluation (*ELL*) phases.

Outline of the chapter: In Section 7.1, we review the challenges and lessons learned related to requirements elicitation. Afterward, Section 7.2 presents how the identified

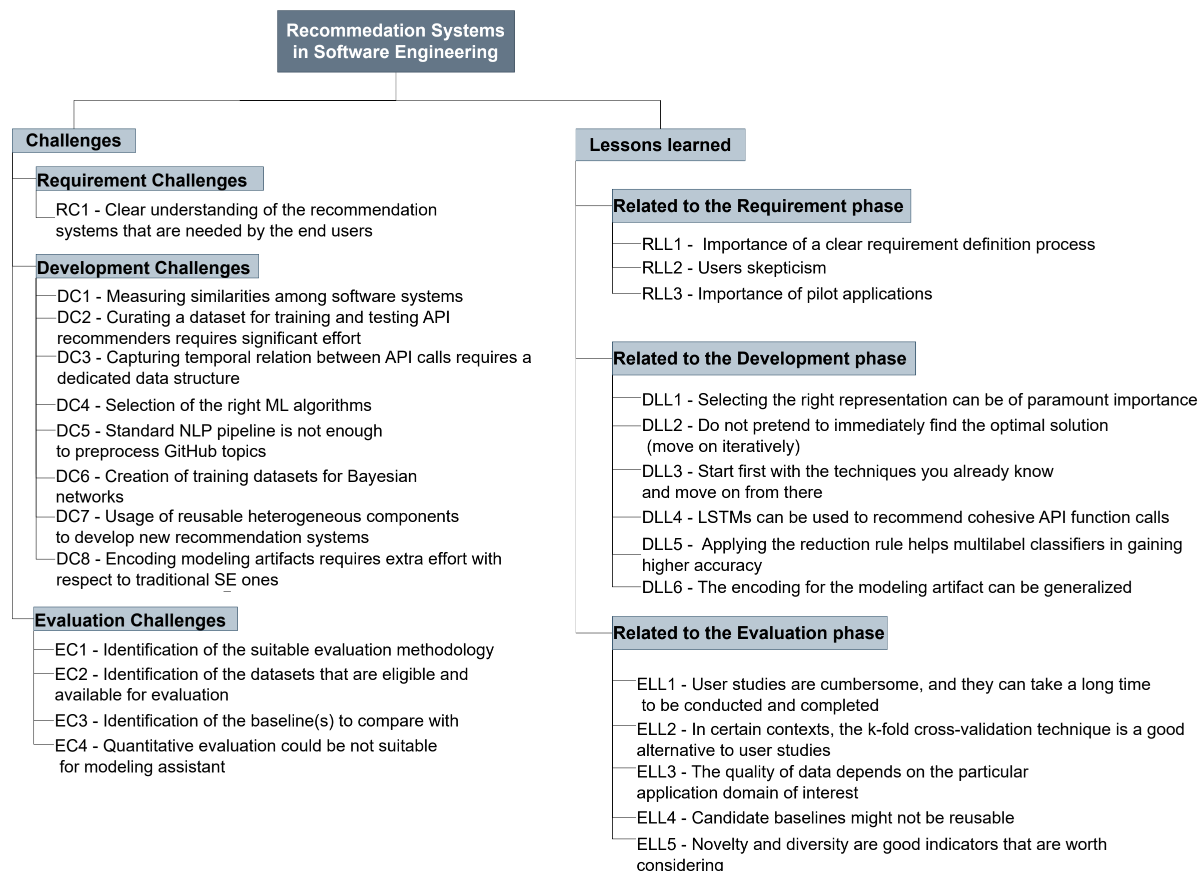


Fig. 7.1 Map of challenges and lessons learned.

challenges have been addressed in CROSSMINER and the lessons learned are discussed in Section 7.3. Finally, we conclude the chapter in Section 7.4 by summarizing the gained experience in recommending heterogeneous software components.

7.1 Challenges and lessons learned related to requirements elicitation

> **RC1 - Clear understanding of the recommender systems that are needed by the end users:** Getting familiar with the functionalities that are expected from the final users of the envisioned recommender systems is a daunting task. We might risk spending time on developing systems that are able to provide recommendations, which instead might not be relevant and in line with the actual user needs.

To deal with such a challenge and thus mitigate the risks of developing systems that might not be in line with the user requirements, we developed proof-of-concept recommender

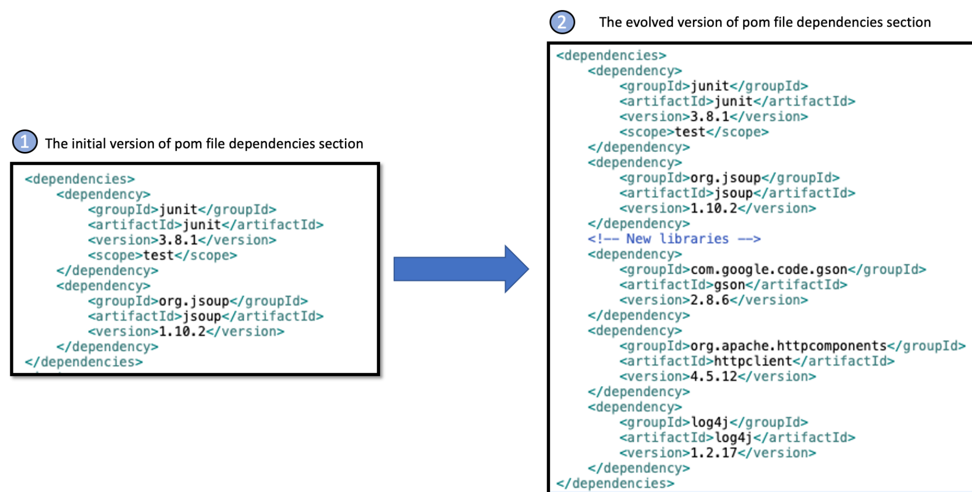


Fig. 7.2 `pom.xml` files of a project before (*left*) and after (*right*) having adopted third-party libraries recommended by CrossRec.

systems. In particular, we implemented demo projects that reflected real-world scenarios in terms of explanatory context inputs and corresponding recommendation items that the envisioned recommender systems should have produced. For instance, concerning CrossRec, we experimented on the *jsoup-example*[1] explanatory Java project for scraping HTML pages. This project consists of source code and few related third-party libraries already included, i.e., *json-soup*[2] and *junit*[3] as shown in the left-hand side of Fig. 7.2.

By considering such project as input, CrossRec provides a list of additional libraries as a suggestion that the project under development should also include. For instance, some beneficial libraries to be recommended are as follows:

(i) gson[4] for manipulating JSON resources; *(ii) httpclient*[5] for client-side authentication, HTTP state management, and HTTP connection management; and *(iii) log4j*[6] to enable logging at runtime. By carefully examining the recommended libraries, we see that they have a positive impact on the project. To be concrete, the usage of the *httpcomponent* library allows the developer to access HTML resources by unloading the result state management and client-server authorization implementation on the library; meanwhile *gson* could provide

[1] https://github.com/MDEGroup/FOCUS-user-evaluation
[2] https://jsoup.org/
[3] https://junit.org/
[4] https://github.com/google/gson
[5] https://hc.apache.org/
[6] https://logging.apache.org/

a parallel way to crawl public Web data; finally introducing a logging library, i.e., *log4j*, can improve the project's maintainability.

Concerning FOCUS, the process was a bit different, i.e., use-case partners were providing us with incomplete source code implementation and their expectations regarding useful recommendations. Such artifacts were used as part of the requirements to implement the system able to resemble them. The use-case partner expects to get code snippets that include suggestions to improve the code, and predictions on next API function calls.

To agree with the use-case partners on the recommendations that were expected from FOCUS, we experimented on a partially implemented method of the *jsoup-example* project named `getScoresFromLivescore` shown in Listing 7.1. The method should be designed so as being able to collect the football scores listed in the livescore.com home page. To this end, a JSON document is initialized with a connection to the site URL in the first line. By using the JSOUP facilities, the list of HTML element of the class `sco` is stored in the variable `score` in the second line. Finally, the third line updates the scores with all of the parents and ancestors of the selected scores elements.

Listing 7.1 Partial implementation of the explanatory `getScoresFromLivescore()` method.

```
public static void getScoresFromLivescore()
throws IOException {
  Document document =
  Jsoup.connect("https://www.livescore.com/").get();
  Elements scores =
  document.getElementsByClass("sco");
  scores = scores.parents();
  ...
}
```

Figure 7.3 depicts few recommendations that our use-case partners expected when we presented the example shown in Listing 7.1. The blue box contains the recommendation for improving the code, i.e., the `userAgent` method is to prevent sites from blocking HTTP requests, and to predict the next *jsoup* invocation. Furthermore, some recommendations could be related to API function calls of a competitor library or extension. For this reason, the green and red boxes contain invocations of *HTMLUnit*,[7] a direct competitor of *jsoup* that includes different browser user agent implementations, and *jsoupcrawler* a custom extension of *jsoup*. FOCUS has been conceptualized to suggest to developers recommendations consisting of a list of API method calls that should be used next. Furthermore, it also recommends real code snippets that can be used as a reference to support developers in finalizing the method

[7] http://htmlunit.sourceforge.net/

0.71711844	org.jsoup.Connection.userAgent(java.lang.String)
0.71711844	org.jsoup.nodes.Document.children()
0.71711844	org.jsoup.nodes.Document.select(java.lang.String)
0.71711844	org.jsoup.nodes.Element.children()
0.71711844	org.jsoup.nodes.Element.select(java.lang.String)
0.71711844	org.jsoup.select.Elements.attr(java.lang.String)
0.71711844	org.jsoup.select.Elements.first()
0.71711844	org.jsoup.select.Elements.html()
0.71711844	org.jsoup.select.Elements.iterator()
0.63971543	com.gargoylesoftware.htmlunit.WebClient.WebClient(com.gargoyles
0.63971543	com.gargoylesoftware.htmlunit.WebClient.close()
0.63971543	com.gargoylesoftware.htmlunit.WebClient.getOptions()
0.63971543	com.gargoylesoftware.htmlunit.WebClient.getPage(java.lang.String)
0.63971543	com.gargoylesoftware.htmlunit.WebClientOptions.setCssEnabled(b
0.63971543	com.gargoylesoftware.htmlunit.WebClientOptions.setJavaScriptEna
0.63971543	com.gargoylesoftware.htmlunit.html.HtmlPage.asXml()
0.63971543	com.snicesoft.jsoupcrawler.config.entity.Info.isCss()
0.63971543	com.snicesoft.jsoupcrawler.config.entity.Info.isJs()
0.63971543	com.snicesoft.jsoupcrawler.core.ParseType.ordinal()

Fig. 7.3 Recommended API calls for the `getScoresFromLivescore()` method in Listing 7.1.

definition under development. More code examples provided by FOCUS are available in an online appendix.[8]

7.1.1 Lessons learned

RLL1 – Importance of a clear requirement definition process: To address Challenge RC1, we applied the requirement definition process shown in Fig. 7.4, which consists of the following steps and that in our opinion can be applied even in contexts that are different from the CROSSMINER one:

- *Requirement elicitation:* The final user identifies use cases that are representative and that identify the functionalities that the wanted recommender systems should implement. By considering such use cases, a list of requirements is produced;
- *Requirement prioritization:* The list of requirements produced in the previous step can be very long, because users tend to add all the wanted and ideal functionalities even those that might be less crucial and important for them. For this reason, it can be useful to give a priority to each requirement in terms of the modalities *shall*, *should*, and *may*. *Shall* is used to denote essential requirements, which are of highest priority for validation of the wanted recommender systems. *Should* is used to denote a requirement that would be not essential even though would make the wanted recommender systems working better. *May* is used to denote requirements that would be interesting to satisfy and explore even though irrelevant for validating the wanted technologies;

[8] https://github.com/crossminer/FOCUS

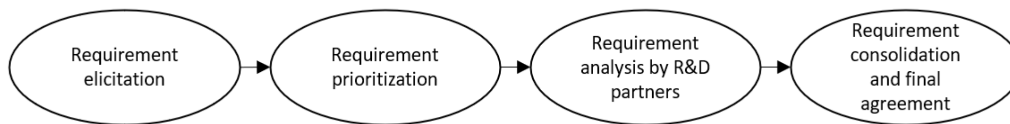


Fig. 7.4 Requirement definition process.

- *Requirement analysis from the existing literature:* The prioritized list of requirements has been extracted from the current literature with the aim of identifying the major components that need to be developed. Possible technological challenges that might compromise the satisfaction of some requirements are identified in this step and considered in the next step of the process;

- *Requirement consolidation and final agreement:* By considering the results of the analysis, the list of requirements is further refined and consolidated. After this step, user case partners have ensured highest priority requirements, which will ended up with the development of the actual RSSEs.

We have applied such a process in different projects and we successfully applied it also for developing the recommender systems that we identified in the context of the CROSSMINER project.

RLL2 – Users skepticism: Especially at the early stages of the wanted recommender systems development, target users might be skeptical about the relevance of the potential items that can be recommended.

We believe that defining the requirements of the desired recommender systems in tight collaboration with the final users is the right way to go. Even when the proposed approach has been evaluated employing adequate metrics, final users still might not be convinced about the retrievable recommendations' relevance. User studies can be one of the possible options to increase the final users' trust, even though a certain level of skepticism might remain when the intended final users have not been involved in the related user studies.

RLL3 – Importance of pilot applications: Using a pilot application can be beneficial to support the interactions between the final users and the developers of the wanted recommender systems.

The application can allow the involved parties to tailor the desired functionalities utilizing explanatory inputs and corresponding recommendations that the envisioned system should produce.

7.2 Challenges and lessons learned related to development

Once each RSSE has been developed, we conduct a careful analysis to identify the most challenging aspect in each phase of the process, i.e., requirement elicitation, actual development, and the outcomes evaluation.

Presenting Recommendations: As the last phase, the produced recommendation items need to be properly presented to the developer. To this end, several strategies involve potentially different technologies, including the development of extensions for IDEs and dedicated Web-based interfaces. `IDEIntegration` offers several advantages, i.e., auto-complete shortcuts and dedicated views showing the recommended items. The integration is usually performed by the development of a plug-in, as shown in existing recommender systems [151, 203]. Nevertheless, developing such an artifact requires much effort, and the integration must take into account possible incompatibilities among all deployed components. A more flexible solution is represented by `WebInterfaces` in which the recommendation system can be used as a stand-alone platform. Even though the setup phase is more accessible rather than the IDE solution, presenting recommendations through a web service must handle some issues, including server connections, and suitable response times. For presentation purposes, interactive data structures might be useful in navigating the recommended items. `TraversableGraph` is just one successful example of this category. Strathcona [112] makes use of this technique to show the snippets of code rather than simply retrieving them as ranked lists. In this way, final users can figure out additional details about the recommended items.

Development challenges for CrossSim and CrossRec In OSS forges like GitHub, there are several connections and interactions, such as development commit to repositories, user star repositories, or projects contain source code files, to mention a few.

DC1 – Measuring similarities among software systems Considering the miscellaneousness of artifacts in open source software repositories, similarity computation becomes more complicated as many artifacts and several cross relationships prevail.

To conceptualize CrossSim [184], we came up with the application of a graph-based representation to capture the semantic features among various actors, and consider their intrinsic connections. We modeled the community of developers together with OSS projects, libraries, source code, and their mutual interactions as an *ecosystem*. In this system, either humans or non-human factors have mutual dependencies and implications on the others. Graphs allow for flexible data integration and facilitates numerous similarity metrics [32].

We decided to adopt a graph-based representation to deal with the project similarity issue because some of the co-authors already addressed a similar problem in the context of Linked Data. The analogy of the two problems inspired us to apply the similarity technique already developed [178] to calculate the similarity of representative software projects. The initial evaluations were encouraging and consequently, we moved on by refining the approach and improving its accuracy.

Despite the need to better support software developers while they are programming, very few works have been conducted concerning the techniques that facilitate the search for suitable third-party libraries from OSS repositories. We designed and implemented CrossRec on top of CrossSim: the graph representation was exploited again to compute similarity among software projects, and to provide inputs for the recommendation engine.

Understanding the features that are relevant for the similarity calculation was a critical task, which required many iterations and evaluations. For instance, at the beginning of the work we were including in the graph encoding information about developers, source code, GitHub star events when available, etc. However, by means of the performed experiments, we discovered that encoding only dependencies and star events is enough to get the best performance of the similarity approach [184].

To sum up, concerning the features shown in Fig. 3.2, both CrossSim and CrossRec make use of a graph-based representation for supporting the *Data Preprocessing* activity. Concerning the *Producing Recommendation* phase, item-based collaborative filtering techniques have been exploited. For the *Capturing Context* phase, the project being developed is encoded in terms of different features, including used third-party libraries, and README files. Recommendations are presented to the user directly in the used Eclipse-based development environment.

Development challenges for FOCUS and LUPE During the development process, rather than programming from scratch, developers look for libraries that implement the desired functionalities and integrate them into their existing projects [180]. For such libraries, API function calls are the entry point which allows one to invoke the offered functionalities. However, in order to exploit a library to implement the required feature, programmers need to consult various sources, e.g., API documentation to see how a specific API instance is utilized in the field. Nevertheless, from these external sources, there are only texts providing generic syntax or simple usage of the API, which may be less relevant to the current development context. In this sense, concrete examples of source code snippets that indicate how specific API function calls are deployed in actual usage, are of great use [167].

Several techniques have been developed to automate the extraction of API usage patterns [219] in order to reduce developers' burden when manually searching these sources and to provide them with high-quality code examples. However, these techniques, based on clustering [193, 276, 298] or predictive modeling [83], still suffer from high redundancy and poor run-time performance.

By referring to the features shown in Fig. 3.2, differently from other existing approaches which normally rely on clustering to find API calls, FOCUS implements a *context-aware collaborative-filtering* system that exploits the cross relationships among different artifacts in OSS projects to represent them in a graph and eventually to predict the inclusion of additional API invocations. Given an active declaration representing the user *context*, we search for prospective invocations from those in similar declarations belonging to comparable projects. Such a phase is made possible by a proper *data preprocessing* technique, which encodes the input data by means of a *tensor*. The main advantage of our tool is that it can recommend real code snippets that match well with the development context. In contrast with several existing approaches, FOCUS does not depend on any specific set of libraries and just needs OSS projects as background data to generate API function calls. More importantly, the system scales well with large datasets using the collaborative-filtering technique that filters out irrelevant items, thus improving efficiency. The produced recommendations are shown to the users directly in the Eclipse-based IDE.

DC2 - Curating a dataset for training and testing API recommenders requires significant effort: To provide input data in the form of a tensor, it is necessary to parse projects for extracting their constituent declarations and invocations.

A major obstacle that we needed to overcome when implementing FOCUS is as follows. To provide input data in the form of a tensor, it was necessary to parse projects to extract their constituent declarations and invocations. However, FOCUS relies on Rascal [25] to function, which in turn works only with compilable Java source code. To this end, we populated training data for the system from two independent sources. First, we curated a set of Maven jar files which were compilable by their nature. Second, we crawled and filtered out to select only GitHub projects that contain informative *.classpath*, which is an essential requirement for running Rascal. Once the tensor has been properly formulated, FOCUS can work on the collected background data, being independent of its origin. One of the considered datasets was initially consisting of 5,147 Java projects retrieved from the Software Heritage archive [64]. To satisfy the baseline constraints, we first restricted the dataset to the list of projects that use at least one of the considered third-party libraries. Then, to comply with the requirements of FOCUS, we restricted the dataset to those projects containing at least one `pom.xml` file. Because of such constraints, we ended up with a dataset consisting of

610 Java projects. Thus, we had to create a dataset ten times bigger than the used one for the evaluation. Concerning LUPE, we feed the system with two different types of software artifacts, i.e., Android and Maven projects. The former has been extracted by mining Android Time machine repositories to collect .apk files. The latter has been extracted by relying on a curated version of Maven dependency graph dataset. Starting from the mined data, we excerpt four different datasets reported in Table 7.1. In particular, we obtain $\mathbf{DS}_2$ and $\mathbf{DS}_{3S}$ by eliciting a subset of projects and invocations from $\mathbf{DS}_1$ and $\mathbf{DS}_3$ respectively. The rationale is two-fold. On one hand, we are interested in analyzing the variation of performance when a smaller amount of data is considered. On the other hand, larger datasets are not suitable to compare LUPE with the considered baselines, i.e., GAPI and FACER.

DC3 - Capturing temporal relation between API calls requires a dedicated data structure : To give *cohesive* recommendations in terms of API, we need to identify the cause-and-effect relationship in the considered API sequences.

The main development challenges concern the encoding of the API sequence to feed the encoder-decoder architecture. In this respect, we mined the Android projects to extract the API function calls and declarations. Afterward, we use a dedicated component to build sequences that preserve the structure of the source code, i.e., Sequence encoder. In particular, we create a vocabulary where each API has a unique numeric ID. In such a way, LUPE creates one-hot vectors that have been used to build matrices that preserves the semantic among the considered API calls.

Development challenges of MNBN and HybridRec In recent years, GitHub has been at the forefront of platforms for storing, analyzing and maintaining the community of OSS projects. To foster the popularity and reachability of their repositories, GitHub users make daily usage of the star voting system as well as forking [35, 121]. These features allow for increasing the popularity of a certain project, even though the search phase has to cope with a huge number of items. To simplify this task, GitHub has introduced in 2017 the concept of *topics*, a list of tags aiming to describe a project in a succinct way. Immediately after the availability of the topics concept, the platform introduced Repo-Topix [84] to assist developers when creating new projects and thus, they have to identify representative topics. Though Repo-Topix is already in place, there are rooms for improvements, e.g., in terms of the coverage of the recommended topics, and of the underpinning analysis techniques. To this end, we proposed MNBN [72], an approach based on a Multinomial Naive Bayesian network technique to automatically recommend topics given the README file(s) of an input repository.

The main challenges related to the development of MNBN concern three main dimensions as follows: *(i) identification of the underpinning algorithm*, *(ii) creation of the training dataset*, and *(iii) usage of heterogeneous reusable complements*, and they are described below.

DC4 – Selection of the right ML algorithms: Due to the well-known no-free lunch theorem that holds for any Machine Learning (ML) approach [286], selecting the suitable model for the problem at hand is one of the possible pitfalls.

Concerning the Machine Learning domain, all relevant results have been obtained through empirical observations undertaken on different assessments. Thus, to better understand the context of the addressed problem we analyzed existing approaches that deal with the issue of text classification using ML models. Among the analyzed tools, the Source Code Classifier (SCC) tool [14] can classify code snippets using the MNB network as the underlying model. In particular, this tool discovers the programming language of each snippet coming from StackOverflow posts. The results show that Bayesian networks outperform other models in the textual analysis task by obtaining 75% of accuracy and success rate. Furthermore, there is a subtle correlation between the Bayesian classifier and the TF-IDF weighting scheme [128]. A comprehensive study has been conducted by comparing TF-IDF with the Supporting Vector Machine (SVM) using different datasets. The study varies the MNB parameters to investigate the impacts of the two mentioned preprocessing techniques for each of them. The evaluation demonstrates that the TF-IDF scheme leads to better prediction performance than the SVM technique. Thus, we decided to adopt the mentioned MNBN configuration considering these two findings *(i)* this model can adequately classify text content and *(ii)* the TF-IDF encoding leads benefits in terms of overall accuracy.

DC5 – Standard NLP pipeline is not enough to preprocess GitHub topics: To increase the accuracy of the examined classifiers, a set of advanced rewriting rules needs to be defined, thus capturing semantic relationships among GitHub topics.

When developing MNBN, we limited ourselves to applying standard NLP pipeline by considering just the stemmed version of featured topics. Nevertheless, this process may produce inaccurate results since the semantics have not been considered. Therefore, we defined a set of rewriting rules built on top of an existing work [120] to increase HybridRec's accuracy. Those rules go a step further with respect to the original work since we considered a different dataset to perform the evaluation. For instance, we expand abbreviations and acronyms that are widely spread in the SE community to make them more consistent. To sum up, the proposed rewriting rules succeeded in increasing the overall performance of HybridRec as reported in the dedicated chapter.

DC6 – Creation of training datasets for Bayesian networks: To make the employed Bayesian network accurate, each topic must be provided with a similar number of training projects; otherwise, the obtained results can be strongly affected by the unbalanced distribution of the considered topics.

To mitigate such issues, we decided to train and evaluate the approach by considering 134 GitHub featured topics. In this respect, we analyzed 13,400 README files by considering 100 repositories for each topic. To collect such artifacts, we needed to be aware of the constraints imposed by the GitHub API, which limit the total number of requests per hour to 5,000 for authenticated users and 60 for unauthorized requests.

DC7 – Usage of reusable heterogeneous components to develop new recommendation systems: The orchestration of heterogeneous components was another challenge related to the development of MNBN.

Though the employed Python libraries are flexible, they involve managing different technical aspects, i.e., handling access to Web resources, textual engineering, and language prediction. Moreover, each component has a well-defined set of input elements that dramatically impact on the outcomes. For instance, the README encoding phase cannot occur without the data provided by the crawler component, which gets data from GitHub. In the same way, the topic prediction component strongly relies on the feature extraction performed by the TF-IDF weighting scheme. Thus, we succeeded in developing MNBN by putting significant efforts in composing all the mentioned components coherently.

To summarize, concerning Fig. 3.2, NLP techniques have been applied to support the data preprocessing phase of MNBN. A model-based approach consisting of a Bayesian network underpins the overall technique to produce recommendations. The user context consists of an input README file, which is mined employing a keyword extraction phase. The produced recommendations are shown to the user directly in the employed Eclipse-based IDE.

Development challenges for MemoRec and MORGAN

DC8 – Encoding modeling artifacts requires extra effort with respect to traditional SE ones: Due to their peculiar nature, modeling artifacts require tailored encoding techniques to extract relevant information.

Since modeling activities involve abstractions of real-world systems, the extraction of relevant information to feed ML-based assistants requires custom techniques that go beyond the ones commonly used in other SE tasks. To address this, MemoRec adopts four encoding schemes to represent different views concerning the terms extracted from packages, classes, and

structural features, thus improving the quality of suggested modeling elements. Furthermore, we developed dedicated parsers to support different modeling artifacts with MORGAN recommendations. In such a way, we are able to encode the underpinning relationships of the considered artifacts, i.e., metamodels, models, and JSON schema.

7.2.1 Lessons learned

This section presents the different lessons learned that have been elicited from the actual development of the presented RSSEs. In particular, we focus on the experiences that are valuable and could be reused in the future whenever we are supposed to run similar projects.

DLL1 – Selecting the right representation can be of paramount importance: A suitable encoding helps capture the intrinsic features of the OSS ecosystem to facilitate the computation of similarities among software projects, moreover it paves the way for various recommendations.

With respect to the features shown in Fig. 3.2, the used graph representation facilitates different recommendations, e.g., FOCUS, MemoRec, and MORGAN making use of a graph-based representation for supporting the *Data Preprocessing* activity. We selected such a representation since some of the co-authors have gained similar experiences in the past and consequently, we followed the intuition to try with the adoption of graphs, and graph-similarity algorithms also in the mining OSS repositories. We started with CrossSim, and subsequently we found that the graph-based representation is also suitable to develop CrossRec and FOCUS. In contrast, GitHub topics preprocessing involved the definition of additional steps, namely the definition of rewriting rules employed by HybridRec. Finally, modeling artifacts required a dedicated process based on well-curated encoding schemes and tailored parsers to properly capture relevant information to feed the corresponding systems.

DLL2 – Do not pretend to immediately find the optimal solution (move on iteratively): Conceiving the right techniques and configurations to satisfy user needs can be a long and iterative process.

For conceiving all the presented recommender systems, we followed an iterative process aiming to find the right underpinning algorithms and configurations to address the considered problems with the expected accuracy. It can be a very strenuous and Carthusian process that might require some stepping back if the used technique gives evidence of inadequacy for the particular problem at hand, fine-tune the used methods, and collect more data both for training and testing. For instance, in the case of CrossSim we had to make four main iterations to identify the features of open source projects relevant for solving the problem of computing

similarities among software projects. During the initial iterations, we encoded more than the necessary metadata. For instance, we empirically noticed that encoding information about developers contributing to projects might reduce the accuracy of the proposed project similarity technique. Similarly, we improved the MNBN overall performance by *i)* adopting a well-curated preprocessing on the GitHub topics and *ii)* considering an enhanced version of the Bayesian model that copes with unbalanced datasets.

DLL3 – Start first with the techniques you already know and move on from there: Starting with low-hang fruits allowed us to gain experience of the domain of interest and to quickly produce even sub-optimal results that can still be encouraging while finding better solutions.

During the development of the proposed RSSEs, we started first with approaching problems that were similar to those we had already in the past. In other words, we first got low-hang fruits and then moved on from them. In this respect, we began early with CrossSim since we noticed some similarities with the problem that one of the co-authors previously addressed [178]. We managed to gain additional expertise and knowledge in the domain of recommender systems, while still satisfying essential requirements elicited from the partners. Afterward, we succeeded in addressing more complicated issues, i.e., recommending third-party libraries with CrossRec [182], API function calls and code snippets with FOCUS [181, 177] and LUPE [], automatic tagging of OSS repositories with MNBN and HybridRec, and supporting the model completion with MemoRec and MORGAN.

DLL4 – LSTMs can be used to recommend cohesive API function calls: Among the examined cutting-edge technologies, LSTMs is capable of preserving the cause-effective relationship that occurs while the APIs have been defined.

By experimenting with different configurations of LUPE, we demonstrate the encoder-decoder architecture is particularly well-suited for recommending API function calls, i.e., this encoding scheme enables the recommendation engine to effectively learn relationships extracted from the source code. Therefore, LUPE is fed with a sequential representation of APIs, thus exhibiting superior prediction performance compared to traditional API recommender systems, e.g., FOCUS. However, the employed LSTM network suffers from performance issues during the training phase. To overcome this, we employed a powerful computation platform that allows us to compute and export the weights that can be used by standard machines.

DLL5 – Applying the reduction rule helps multilabel classifiers in gaining higher accuracy: The rewriting rules employed by HybridRec increase the overall performances, especially in terms of coverage.

The conceived rewriting rules have been carefully crafted and tailored by considering the underlying semantic relationships among GitHub topics. By improving existing work, the application of these customized rewriting rules results in a significant improvement in terms of catalog coverage, meaning that HybridRec is capable of retrieving more diverse recommendations compared to our previous approaches, i.e., MNBN and TopFilter. Therefore, it is our strong belief that embodying semantics in a proper structure can improve the automatic OSS classification.

DLL6 – The encoding for the modeling artifact can be generalized: Although modeling are heterogeneous in most of the cases, adopting dedicated parsers can be used to extract the needed information.

While developing MORGAN, we implemented three different parsers specifically conceived for each type of artifact. Even though the kind of information is strictly bounded by the application domain, we preserve the same structure to preserve the same structure, with the aim of enabling the graph kernel similarity adopted by the approach. While this is not enough to claim the generalizability of the approach, this is an initial step towards the integration of modeling assistants that can handle heterogeneous modeling artifacts.

7.3 Challenges and lessons learned related to evaluation

Once the recommender systems had been realized, it was necessary to compare them with existing state-of-the-art techniques. Evaluating a recommendation system is a challenging task since it involves identifying different factors. In particular, there is no golden rule for evaluating all possible recommender systems due to their intrinsic features as well as heterogeneity. To evaluate a new system, various questions need to be answered, as they are listed as follows:

– *Which evaluation methodology is suitable?* Assessing RSSE can be done in different ways. Conducting a user study has been accepted as the *de facto* method to analyze the outcome of a recommendation process by several studies [158, 167, 203, 295, 298]. However, user studies are cumbersome, and they may take a long time to finish. Furthermore, the quality of a user study's outcome depends very much on the participants' expertise and willingness to participate. In this sense, setting up an automated evaluation, in which the manual intervention is not required (or preferably limited), is greatly helpful.

– *Which metric(s) can be used?* Choosing suitable metrics accounts for an important part of the whole evaluation process. While accuracy metrics, such as *success rate*,

precision and *recall* have been widely used to measure the prediction performance, we suppose that additional metrics should be incorporated into the evaluation [90, 182], aiming to study RSSE better. In the scope of this dissertation, we refer to the set of metrics discussed in Chapter 3.

– *How to prepare/identify datasets for the evaluation?* One needs to take into account different parameters when it comes to choosing a dataset for evaluation. Moreover, the data used to evaluate a system depends very much on the underpinning algorithms. In this sense, advanced techniques and methods for curating suitable data are highly desirable.

– *What could be a representative baseline for comparison?* To show the features of a new conceived tool and give evidence of its novelty and advantages, it is necessary to compare it with existing approaches with similar characteristics. Since the solution space is vast, comparing and evaluating candidate approaches can be a daunting task.

EC1 – Identification of the suitable evaluation methodolgy: Deciding the evaluation methodolgy to be applied has to take into account several aspects including the time and efforts that have been allocated for such a phase in the project the work is contextualized.

User studies can be done as *field studies* or as *controlled experiments*. By the former, the participants with different programming experience levels have to complete a list of tasks using the proposed recommendation system without any intervention. The latter is conducted in a monitored environment, and the assigned tasks are carefully tailored for specific purposes. Although these strategies produce remarkable results in various work, there are some issues to be tackled. Among others, the selection of the participants has a crucial role to play.

It is worth noting that the selection of ground-truth data from an active project impacts the evaluation, and it might jeopardize the integrity of the evaluation process. Different aspects, i.e., the scope of the recommendation, the recommendation input, the size of the ground truth, and the characteristics of the selected objects, should be carefully considered to mimic a real usage scenario when it comes to an automatized evaluation. For instance, randomly choosing the ground truth size and objects does not guarantee that the evaluation mimics a real usage scenario. The ground truth extraction strategies that have been employed for evaluating the developed recommender systems are explained below.

CrossRec: Given a set of libraries that an active project uses, CrossRec returns a set of additional libraries that similar projects to the active project have also included. For this reason, in the CrossRec evaluation process, given an active project, a half of its libraries are used as the ground-truth, and the remaining are used as the query. In this case, we split randomly into such sets the libraries that an active project includes.

FOCUS and LUPE: Given a list of method declaration and method invocations pairs, and an active method context, FOCUS predicts the next method invocations that can be added to the active declaration. To simulate a developer's behaviour at different stages of a development project, we performed various evaluation experiments by varying the size of the recommendation query and the size of the ground-truth data. In particular, four different configurations have been considered in the evaluation to mimic the following scenarios:

- the developer is at an early stage of the development process, and the active method is almost empty;
- the developer is at an early stage of the development process, and the active method implementation is well defined;
- the developer is near to the end of the development process, and the active method is almost empty;
- the project is in an advanced development phase, and the active method implementation is well defined.

The ground truth data is extracted accordingly to the scenario that the evaluation mimics.

A similar process has been conducted to evaluate LUPE by splitting the developer's context according to different threshold. Furthermore, we consider a further configuration where only definitions that rarely appear have been considered for the testing phase.

MNBN and HybridRec: Given an active project, the two recommendation system uses the content of *README* file(s) to recommend relevant GitHub topics. Since the recommendation input does not coincide with the object of the recommendation, we used the whole list of topics that an active project uses as ground-truth. It is worth noting that for the MNBN we consider just the featured topics while HybridRec has been assessed by considering the whole set of tags filtered by using the mining rules.

MemoRec and MORGAN: Due to the lack of proper integration with a modeling environment, we adopt the same strategy to evaluate MemoRec and MORGAN. The former has been evaluated on a curated dataset composed of metamodels by adopting the ten-fold cross-validation. Similarly, the performance of the latter has been assessed by enlarging the application domain, by considering five different datasets.

It is our firm belief that user studies are inevitable in many contexts. For the evaluation of CrossSim [183], a user study is a must, since there are no other ways to evaluate the similarity between two OSS repositories, rather than the manual scoring done by humans. We may avoid user studies in some specific cases. For instance, when evaluating CrossRec [182],

we realized that with the application of the ten-fold cross-validation technique, we can rely on the available data to perform the evaluation, without resorting to a user study. For FOCUS [181, 177], while we can use data to evaluate its performance, we assume that its usability and usefulness can be properly studied only with a user study, where developers are asked to give their opinion on a specific API call recommended by the system.

EC2 – Identification of the datasets eligible and available for evaluation: The datasets used for evaluation depend very much on the recommendation algorithms being used.

For each developed tool, we had to go through the following dimensions related to datasets:

- *Which format?* Depending on the employed recommendation techniques (e.g., collaborative filtering, LSTMs, graph kernels etc.) we had to identify the proper ways to encode the created datasets. For instance, to enable the application of a graph-based similarity algorithm underpinning CrossSim, we had to encode the different features of OSS projects in a graph-based representation. The same datasets needed to be represented in a TF-IDF format to enable the application of FOCUS;

- *Which preprocessing process should be applied to create the dataset?* To minimize the size of the input datasets and thus to make their manipulation efficient, we had to perform different data filtering tasks. For instance, in the case of CrossSim, to enable the application of the employed graph-similarity algorithm, we identified the features that are relevant for the task. For example, information about software developers, source code, and GitHub topics was filtered out from the available datasets even though it was easy to encode all of them as elements in the input graphs. Similar data filtering phases were also performed in CrossRec to enable the recommendation of third-party libraries that might be added in the project under development. Indeed, such data filtering phases have to be performed without compromising the performance (in terms of accuracy, precision, recall, etc.) of the approach under evaluation;

- *Which limitations should we tackle when collecting the dataset?* The primary limitations we experienced when evaluating recommender systems were related to the GitHub APIs restrictions. Unfortunately, the adoption of alternative sources like GHTorrent [96] was not enough due to the lack of needed artifacts such as source code. Knowing such limitations in advance, when collecting projects from GitHub, we decided to save as much data as possible for every single project. The goal was to enable the reuse of the collected data even for perspective evaluations to be done for future recommender systems to be developed in the SE domain.

EC3 – Identification of the baseline(s) to compare with: To showcase the features of a new conceived tool as well as to demonstrate its novelty, it is necessary to compare it with existing approaches with similar features. In fact, choosing the correct baseline is a challenging activity, as to ensure a fair comparison, the baselines to be considered must be endowed with reusable *tools* and *datasets*.

While in general, authors of the selected baselines published their tools and dataset online, it is the case that many of them are faulty, or not well maintained, or even worse: no longer available. In particular, while developing the presented RSSEs we always tried our best to identify the baselines to be used for the evaluations. Unfortunately, often they were not available, which indeed led to difficulties in the evaluation. For instance, for evaluating CrossSim, since the implementations of the baselines were no longer available for public use, we had to re-implement them by strictly following the descriptions in the original papers [88, 158, 295]. That was not possible for evaluating MNBN due to the lack of details in the publicly available documents describing the corresponding baseline. In general, whenever a baseline is selected, and it is not available online, we contacted its authors for the original implementation. It was rare that we got a response from the authors with the tool and/or data. Thus, for the particular cases of the developed recommender systems, either we re-implemented the tool, as it is the case with CrossSim, or we compared by performing experiments on the datasets that have been used in the original papers, as we did for CrossRec.

Table 7.1 RSSEs evaluation facts.

		CrossSim [175]	CrossRec [182]	FOCUS [181, 177]	LUPE [190]	MNBN[72]	HybridRec [70]	MemoRec [69]	MORGAN [76]
Methodology	Cross-Val.		✓	✓	✓	✓	✓	✓	✓
	User study	✓		✓					
Metric	Success rate	✓				✓	✓	✓	✓
	Precision	✓	✓	✓	✓	✓	✓	✓	✓
	Recall		✓	✓	✓	✓	✓	✓	✓
	F-score		✓	✓	✓	✓	✓	✓	✓
	nDCG		✓						
	TopRank					✓	✓		
	Levenshtein			✓	✓				
	Coverage		✓				✓		
	Entropy		✓						
	Novelty		✓						
	Confidence	✓							
	Ranking	✓							
	Time	✓		✓	✓			✓	✓
Dataset	Source	GitHub	GitHub	GitHub, Maven central repository	Maven central reposiotry,Android Time Machine	GitHub	GitHub, Maven central repository	Babur [300]	Babur [300], ModelSet [149], mined Java models, mined JSON schema
	Size	580 projects	1,200 projects	3,600 projects	9,688 projects	13,400 projects	26,499 projects	555 metamodels	6,604 modeling artifacts
Artifact		Metadata	Metadata	Source code	Source code	README files	README files	Metamodels	Metamodels, Models, and JSON schema
Baseline		MUDABlue [88], CLAN [158], RepoPal [295]	LibRec [259], LibFinder [197], LibCUP [229]	UP-Miner [276], PAM [83]	GAPI [145], FACER [9]	None	None	None	None

Table 7.1 summarizes the main factors related to the evaluation of our proposed recommendation systems. Depending on the intrinsic characteristics of each tool, different metrics and methodologies were employed to evaluate them. For example, to study CrossSim [175, 183], a user study with several developers' involvement was the only option since there is no automated method to evaluate the similarity between two OSS projects. Meanwhile, with CrossRec [182], FOCUS [181], and MNBN [72], we relied only on data to investigate their performance. Moreover, depending on the availability of baselines and quality requirements, we used different evaluation metrics, such as Accuracy (Precision, Recall, TopRank) or Sales Diversity (Coverage, Entropy). Choosing suitable data plays an important role in the evaluations, and it depends on various factors, such as systems' characteristics, baselines, evaluation purposes, or even constraints imposed by OSS platforms, e.g., GitHub and the Maven central repository. The selection of baselines was also a significant issue, considering their complexity and relevance with our tools. For evaluating CrossRec, we were able to consider three different tools for comparison, i.e., LibRec [259], LibFinder [197], and LibCUP [229]. While with FOCUS, only PAM [83] was selected to benchmark since other relevant tools such as MAPO [298] and UP-Miner [276] were no longer available. In summary, we believe that there are many factors when it comes to designing and evaluating a recommendation system, and we should carefully investigate the most probable scenarios to select the optimal one.

EC4 – Quantitative evaluation could be not suitable for modeling assistant: To evaluate modeling assistants, the widely adopted process involves mimicking the modelers' behavior by using automatic techniques. Even though we can measure the overall accuracy, such strategies cannot be used to evaluate the system qualitatively.

Different from other tasks that can be automated using algorithms or machine learning techniques, modeling requires the expertise and understanding of domain knowledge by modelers. Therefore, automating assessment, even if it is well-conducted, could lead to erroneous claims on the accuracy of the system. Concerning the developed systems, namely MemoRec and MORGAN, we did not have the possibility of including a user study since the first objective is to support model completion rather than the full integration in a modeling environment. On one hand, the two approaches achieve low accuracy in terms of the examined metric. On the other hand, designing proper user studies for modeling assistants is still an open challenge due to the development effort required to integrate those systems in a stand-alone environment.

7.3.1 Lessons learned

ELL1 – User studies are cumbersome, and they can take a long time to be conducted and completed: The quality of a user study's outcome depends very much on the participants' expertise and willingness to participate.

People are often not very keen on the experiments, since there is no incentive/reward for performing the required tasks. Moreover, there is a trade-off between domain-expert developers, who may not need a recommendation system to develop, and students who have never used this type of system. As a result, we evaluated CrossSim by involving 15 developers of different background of knowledge. Aiming at a reliable evaluation, for each query we mixed and shuffled the top-5 results generated from the computation by each similarity metric in a single Google form and presented them to the evaluators who then inspected and given a score to every pair. Thus, we managed to mimic a *taste test* where users are asked to judge a product, e.g., food or drink, without having a priori knowledge about what is being evaluated [92, 201]. In this way, we removed any bias or prejudice against a specific similarity metric. The participants were asked to label the similarity for each pair of query and retrieved project regarding their application domains and functionalities. Furthermore, we also allowed for cross-checking, i.e., the results of one developer were validated by the others. To perform such evaluation for CrossSim and compare it with the baselines, it has been crucial to design the experimental settings properly and clearly define the manual evaluation tasks by adhering to the taste-test methodology.

ELL2 – In some certain contexts, the k-fold cross-validation technique is a good alternative to user studies: We realized that user studies are cumbersome, and they can take a long time to conduct and complete. However, we experienced that the assessment can also be automatized by means of case studies or data itself.

By the former, use cases are pre-selected for the recommendation. By the latter, we set up an automated evaluation, in which the manual intervention is not required, or preferably limited. Depending on the availability of data, we managed to avoid performing user studies by employing the *k-fold cross validation* technique [287], which has been popularly chosen as the evaluation method for a model in Machine Learning. By this method, a dataset is divided into *k* equal parts (*folds*). For each validation round, one fold is used as a testing and the remaining *k-1* folds are used as training data. Such an evaluation attempts to mimic a real scenario: *the system should produce recommendations for a project based on the data available from a set of existing projects*. The artifact being considered as the recommendation target is called *object*. For instance, regarding third-party libraries recommendation [182, 259], objects are libraries that a system provides as its outcome. It

is essential to study if the recommendation system is useful by providing the active project with relevant libraries, exploiting the training data. To this end, we keep a certain amount of objects for each active project and use them as input for the recommendation engine, which can be understood as the query. The rest is taken out and used as *ground-truth data*. The ground-truth data is compared with the recommendation outcomes to validate the system's performance. It is expected that the recommendation system can retrieve objects that match up the ones stored as ground-truth data.

ELL3 – The quality of data depends on the particular application domain of interest: While developing the different systems, we further confirmed the importance of having the availability of big data and high-quality data for training and evaluation activities.

The definition of data quality cannot be given in general, and it very much depends on the particular application of interest. According to our experience, creating a dataset, which can be rightly used for both training and evaluating the developed recommender systems, can require significant effort, which can be comparable to that needed to realize the conceived approach. For instance, to implement MNBN, we devoted a huge effort to create the dataset that was supposed to be balanced with respect to the considered GitHub featured topics. Moreover, it can be challenging to collect big datasets, especially when there are several constraints to be satisfied. For instance, in the case of the FOCUS evaluation, one of the considered datasets was initially consisting of 5,147 Java projects retrieved from the Software Heritage archive [64]. To comply with the requirements of the baseline, we first restricted the dataset to the list of projects that use at least one of the considered third-party libraries. Then, to comply with the requirements of FOCUS, we restricted the dataset to those projects containing at least one `pom.xml` file. Because of such constraints, we ended up with a dataset consisting of 610 Java projects. Thus, we had to create a dataset ten times bigger than the used one for the evaluation.

ELL4 – Candidate baselines might not be reusable: When conceiving new recommender systems there can be no baselines to compare with.

There are at least two motivations: *(i)* the proposed approach is the first attempt dealing with the considered problem; *(ii)* the tools and datasets of existing baselines are no longer available or reusable. In such cases, according to the facts shown in Table 7.1, the k-fold cross-evaluation has been a valuable technique that allowed us to evaluate most of the proposed recommender systems even when the baselines were not available. Concerning CrossSim, we decided to perform a user study, to mitigate any bias related to the fact that we re-implemented all the baselines.

ELL5: Novelty and diversity are good indicators that are worth considering: Besides the accuracy, an RSSE should provide relevant items for the active context rather than the most popular.

Many existing approaches just choose to recommend *popular* items, e.g., MUSE [167], PROMPTER [203], LibRec [259]. Through the evaluation of CrossRec, we showed that further than popularity, *novelty* and *diversity* are good indicators for assessing if the recommendation outcomes are meaningful. Among others, the ability to recommend items in the long tail is essential: we can suggest things even when extremely unpopular since a small number of projects use each. However, they turn out to be useful as all of them match those stored as ground-truth. This implies that the novelty of a ranked list is important: a system should recommend libraries that are *novel* [48], i.e., those that have been rarely seen. In this sense, we see that CrossRec can produce good outcomes, not only in terms of success rate and accuracy but also *sales diversity* and *novelty*. Moreover, *serendipity* has been widely exploited to evaluate recommender systems in other domains. Serendipity means that items are obtained by chance but turn out to be useful. However, it seems that the metric has been neglected in evaluating RSSEs. Investigating the importance of serendipity in the context of source code/library recommendation can be an interesting topic. For example, a recommendation engine provides an artifact, e.g., a third-party library or an API function call, which does not belong to the ground-truth data at all; however, it is indeed useful for the current project.

7.4 Conclusion

This chapter presented the challenges and lessons learned on various aspects investigated during the development of the presented recommender systems. We attempt to share with the community the main challenges we had to overcome as well as the corresponding lessons during the three different phases to build and evaluate recommender systems, i.e., requirement, development, and evaluation.

Being focused on heterogeneous recommender systems allowed us to garner many useful experiences and learn important lessons. In the first place, the process yields up a list of actionable items when designing and implementing recommender systems, namely: *(i)* the skepticism that final users can have especially at the early stages of the development and usage of the proposed recommender systems; *(ii)* difficulties in retrieving and creating datasets to be used both for training and evaluation purposes; *(iii)* criticalities related to the selection of baselines for evaluation especially when the related tools are no longer available;

(iv) the variety of evaluation approaches and metrics that can be employed to assess the strengths and limitations of the conceived recommender systems.

Chapter 8

An MDE-based methodology to engineering recommender systems

So far, we presented a set of RSSEs that succeeded in supporting various SE tasks. Despite this, their development is a daunting task that solicits a deep knowledge of various technologies and tools, such as the algorithms to be deployed, the evaluation protocols to be adopted, and the metrics to be considered to assess recommendation accuracies. In recent years, several approaches have been conceived by academia, and industry [79, 85, 30] to facilitate the selection of suitable techniques, aiming to decrease the burden related to the development and evaluation of RSs. However, though the existing tools represent relevant facilitators for the reproducibility of experiments, they still involve performing development activities with the programming languages that were used to build the selected frameworks. Therefore, despite the availability of many algorithms and evaluation techniques, their adoption remains an issue for those who do not have enough expertise or programming skills to develop their own RSs.

To cope with the aforementioned issues, this chapter present LEV4REC, an MDE-based tool to assist developers in designing, configuring, and delivering recommender systems by taking inspiration from the low-code paradigm [73]. Given an initial configuration specified by the user, the system can realize the designed components by generating the corresponding source code implementation. LEV4REC provides users with an environment *(i)* to select the components that need to be used for the wanted recommendation system; *(ii)* to configure the chosen modules through a dedicated modeling environment. The two specifications conform to a dedicated metamodel defined using the Eclipse Modeling Framework (EMF) [100]. In addition, a generator module built on top of Acceleo[1] can generate Python source

[1]https://www.eclipse.org/acceleo/

code implementing the specified RS automatically. In such a way, LEV4REC allows RS developers to define, fine-tune, and test their recommender systems. To validate our approach, we deployed LEV4REC to build two real-world RSs. From an empirical evaluation of various datasets, we see that our conceived environment can adequately define and implement the critical components of the considered systems, allowing them to follow their original design and implementation.

Furthermore, we deploy LEV4REC as *i)* a web-based Xtext editor and *ii)* a plug-in for two different IDEs, i.e., VS code[2] and Eclipse.[3] This aims to demonstrate the tool's capability of generating different systems using the elicited concepts even though the case study evaluation suffer from some limitation, i.e., automated evaluations are not suitable to investigate qualitative aspects such as usability or extensibility. Therefore, we carried out the well-founded focus group methodology[4] to collect feedback from five experts in various domains, including model-driven engineering, recommender systems, and low-code engineering.

The candidate fully contributes to this chapter, from the design to the actual development of the tool. The foundational aspects and the first tool prototype of this work have been published as a workshop paper [71] and a tool demonstration [73] respectively.

Outline of the chapter: Section 8.1 presents the supporting technologies that have been used to develop LEV4REC. The tool is described in Section 8.2, including the two dedicated RS model artifacts, and the integration within IDEs. Two different use case and the evaluation strategies adopted to assess the tool are shown in Section 8.3 and Section 8.4 respectively. We report the obtained results in Section 8.5 and the tool's limitations in Section 8.6. Section 8.7 concludes the chapter by envisioning possible future works in the domain.

8.1 Supporting technologies

Feature-Oriented Software Development (FOSD) [256] is a design paradigm for implementing software systems based on a predefined set of features, e.g., characteristics or requirements. In particular, FOSD aims to modularize a software system by identifying and combining feature modules. *Feature model* [255] is a core concept in FOSD consisting of a set of organized features and constraints between them, e.g., if a specific feature is selected while some others cannot be. Such a model is defined by a *feature tree* that is a hierarchical

[2] https://code.visualstudio.com/
[3] https://www.eclipse.org/
[4] https://psd.ca.uky.edu/files/focus.pdf

representation of features. These features are composed in different ways, e.g., they may be mandatory, optional, or mutually exclusive depending on the ones already selected. Several valid *feature configurations* can be chosen to satisfy the constraints defined at the level of the feature model, depending on the input features.

Possible applications of feature models to configure recommender systems and machine learning applications have been analyzed in [82]. The study evaluates the feasibility of feature models under three dimensions, i.e., interactive configuration, reconfiguration, and modeling processes. Besides the analyzed applications, this study highlights other scenarios, e.g., modeling user interactions, intelligent grouping features, and constraints, to name a few. In recent work, a dedicated feature model has been used to elicit essential components for recommender systems in MDE [12]. To understand which modeling tasks are susceptible to recommendations, the authors conduct a systematic mapping study by analyzing 66 papers to foster future research in this domain.

Concerning the development of recommender systems, several frameworks and libraries have been released in recent years. The RankSys tool [267] supports the rapid prototyping and testing of RSs based on the collaborative-filtering technique. Similarly, the Surprise library [116] offers a collection of algorithms specifically designed for rating prediction. Furthermore, it also provides utilities to evaluate the performance of each implemented algorithm. Besides traditional techniques, Machine Learning (ML) models have been widely used to build recommender systems [246]. Among others, ML frameworks such as Scikit-learn,[5] PyTorch,[6] TensorFlow,[7] or Keras[8] provide a rich set of functionalities, which enables developers to flexibly customize their implementation by fine-tuning hyperparameters.

8.2 The LEV4REC environment

This section presents LEV4REC, an extensible environment for supporting the development of recommender systems. The proposed approach relies on the assumption that it is possible to identify typical components building up RSs and recurrent stages underpinning their evaluation [19]. In this respect, by adhering to the principles and methods supporting the development of recommender systems [33, 71], we identified a set of features characterizing any RS, and proposed the tool-supported process shown in Fig. 8.1.

According to the proposed methodology, an RS developer (RSD) starts with the *RS Feature Selection* phase to select the features of the RS under development, e.g., the rec-

[5] https://scikit-learn.org/
[6] https://pytorch.org/
[7] https://www.tensorflow.org/
[8] https://keras.io/

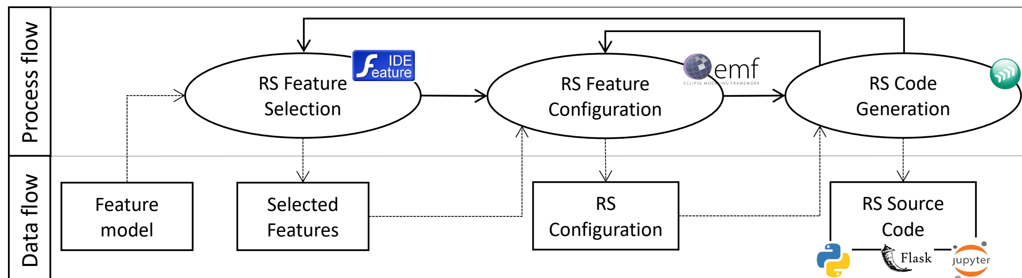


Fig. 8.1 Overview of the proposed approach.

ommendation algorithm to be employed, the preprocessing phases to be operated on the selected dataset, and the evaluation procedure to be performed. A dedicated set of constraints supports such a selection. For instance, if the designer decides to use a supervised dataset, the environment automatically disables the possibility of selecting algorithms for unsupervised learning. After the selection of the wanted features, the environment supports the subsequent *RS Configuration* phase. For instance, if in *RS Feature Selection* the user has selected cross-validation as an evaluation technique to be employed, during the *RS Feature Configuration* phase, the wanted number of folds needs to be specified. The final step of the process consists of the *RS Code Generation* activity. It takes as input the complete *RS Configuration* and generates the source code of the specified RS. Python, Flask,[9] and Jupyter[10] are the target technologies that are currently supported. The proposed approach facilitates the possibility of testing different features and configurations as shown in Fig. 8.1 by the arrows from *RS Code Generation* back to *RS Feature Selection* and *RS Feature Configuration*. We detail each phase of the proposed process in the following subsections.

8.2.1 RS Feature Selection

By carefully review existing technologies, we defined an agnostic feature model to specify different types of recommender systems according to the developer's needs. As shown in Fig. 8.2, the specified model has two main elements defined as follows.[11]

- **Recommender component**: This feature includes all the necessary building blocks to construct an RS, e.g., preprocessing techniques, datasets, algorithms, and evaluation utilities, to name a few. Different subfeatures for each principal component have

[9] https://flask.palletsprojects.com/
[10] https://jupyter-notebook.readthedocs.io/en/stable/notebook.html
[11] For the sake of clarity, only some of the features are shown. Folded features are annotated in Fig. 8.2 with an integer representing the number of the sub-features that are hidden. The interested reader can refer to the complete feature model, which is available at https://tinyurl.com/5yxawyfe.

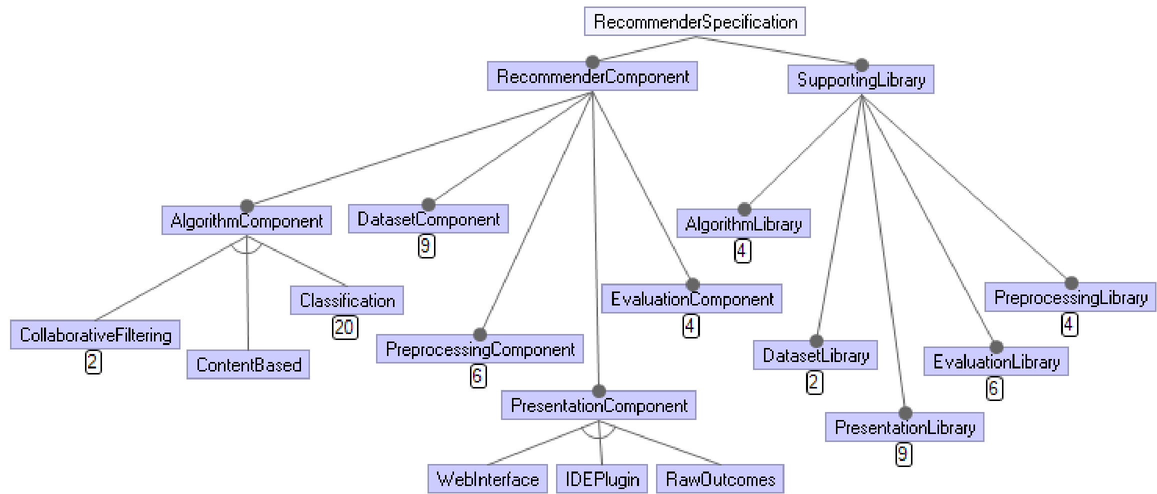


Fig. 8.2 Overview of the proposed feature model.

been defined to cover the desired RS functionalities. For instance, depending on the initial requirements considered during the recommendation process, it is possible to select a collaborative-filtering algorithm on top of either the user-based or item-based technique [125].

- **Supporting library**: Alongside the system design, the developer needs to specify the software libraries that can be adopted to support the specified recommender components. For instance, if the user selects the *ContentBased* algorithm component, the libraries that do not support this algorithm are disabled by leaving only *Surprise* as a possible selection. The feature sub-trees also deal with potential incompatibilities among different libraries, for instance, when the developer cannot apply a data splitting technique provided by the Scikit-learn library on a Surprise algorithm.

As previously mentioned, the whole feature selection phase is restricted through a set of defined constraints that can either include or exclude certain features depending on the current selection. Table 8.1 reports and explains the set of the constraints embedded in the LEV4REC feature model. For instance, a *Supervised algorithm* cannot be selected if the approach relies on a *Unsupervised Dataset* to produce recommendations. Similarly, we impose constraints that act on the software libraries employed to generate the actual RS code. In particular, the *Surprise* feature dictates that the *Surprise preprocessing* and *Surprise Split* features must be adopted to work appropriately by excluding other libraries, e.g., Scikit-learn.

Table 8.1 Constraint-supported RS specification.

Constraint	Description
¬ Scikit-learn ⟹ ¬ Seaborn	The *Scikit-learn* library is required to run the *Seaborn* graphical utilities
¬ TextualData ⟹ ¬ NLP	As the *NLP* pipeline acts on textual data, it requires such a format to be enabled
CollaborativeFiltering ⟹ ¬ Scikit-learn ∧ ¬ PyTorch ∧ ¬ TensorFlow	Selecting *Collaborative filtering* technique automatically excludes software libraries that do not support it, e.g., Scikit-learn, TensorFlow, or PyTorch
IDEPlugin ⟹ IDELibrary	If the *IDEPlugin* feature is selected then *IDELibrary* must be set as the presentation layer generator
RandomSplitting ⟹ SKRandomSplit ∨ SurpriseRandomSplit	If *CrossFold* uses a random splitting rule, RSD can select one of the supported libraries, i.e., *SKRandomSplit* or *SurpriseRandomSplit*
Sklearn ⟹ SklearnPreprocessing ∧ SklearnSplit	To preserve consistency, the system automatically selects *SklearnPreprocessing* and *SklearnSplit* for the corresponding library
SplittingKfold ⟹ SurpriseCrossFold ∨ SKCrossFold	Similar to the *RandomSplitting* feature, RSD can select two different *CrossFold* implementations depending on the chosen library
SupervisedDataset ⟹ ¬ UnsupervisedAlgorithm	The system selects automatically an *UnsupervisedAlgorithm* if RSD specifies a *SupervisedDataset*
Surprise ⟹ SurpriseSplit ∧ SurprisePreprocessing	If *Surprise* is selected as AlgorithmLibrary, then the preprocessing and splitting policies must belong to this library
UnsupervisedDataset ⟹ ¬ SupervisedAlgorithm	If *Unsupervised Dataset* is selected then the developer cannot select supervised algorithms
WebInterface ⟹ WebLibrary	*WebLibrary* must be used as the presentation layer if the feature *WebInterface* is set

8.2.2 RS Feature Configuration

Once the feature model has been defined, the next step is the generation of an initial RS configuration conforming to the metamodel shown in Fig. 8.3. The metamodel defines all the constructs that can be used to define a complete RS configuration, which is then used to generate the source code of the designed RS. The root element named RSModel is composed of several abstract components depicted in light grey, i.e., Dataset, Recommender system, Validation Technique, and PresentationLayer. Each entity can be specialized with concrete elements needed to implement different functionalities. The metaclasses extending such abtract elements are described in the following.

At the beginning of the design process, the user must specify the type of data that will be used to feed the recommender system. Figure 8.4 depicts Dataset specializations which come in handy by providing the information needed to represent a dataset, i.e., independent variables, preprocessing techniques, e.g., Natural Language Preprocessing, normalization, ifnextchar.etcetc.., and the corresponding datasetManipulationLibrary that is

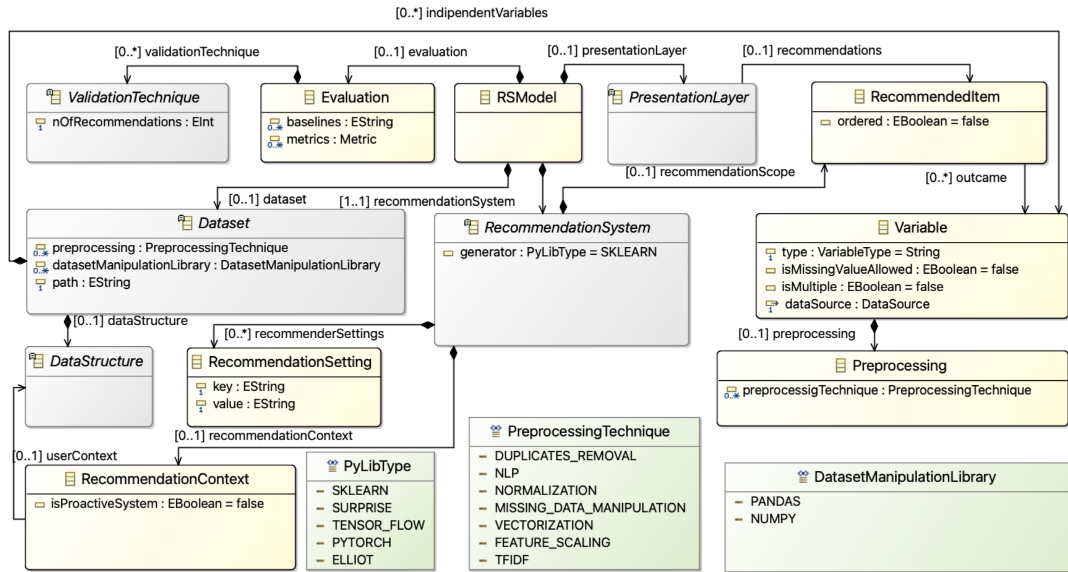


Fig. 8.3 The LEV4REC configuration metamodel.

used to manipulate the datasets at hand, e.g., *pandas* [156] and *numpy* [264]. Since recommender systems rely on heterogenous data, LEV4REC provides `Data Source` concepts designed to collect and organize information coming from differerent sources, i.e., `Code repository`, `Communication Channel`, and `Bug Tracking`. Furthermore, the `DataStructure` metaclass is used to dictate the dataset features, e.g., the format, the size, to name a few. For instance, data extracted from a given GitHub repository could be organized in a `Graph` as well as a `Matrix` depending on the nature of the recommender system. In other words, the system is capable of generating a tailored preprocessing module by relying on the meta-concepts expressed in the `Dataset` metaclass.

Once the dataset has been specified, the user can configure the underpinning recommendation algorithm and its internal parameters by using the `RecommendationSystem` concepts shown in Fig. 8.5. Depending on the `RecommendationContext`, several algorithms can be chosen, e.g., `MachineLearningBasedRS`, `FilteringRS`, to name a few. The user can also implement a `CustomRecommender` if none of the implemented systems can offer the desired functionalities. Afterward, the elicited algorithm can be decorated with different parameters using `RecommendationSetting` and several enumerations entities represented by the green boxes. For instance, if a filtering algorithm is selected, it is possible to specify the similarity function and the type of the algorithm among the available ones.

The designed algorithm eventually produces a list of `RecommendedItem` that can be used by the next components, i.e., `ValidationTechnique` and `PresenationLayer`, to evaluate the system and deliver the outcomes to the user, respectively. The former specifies

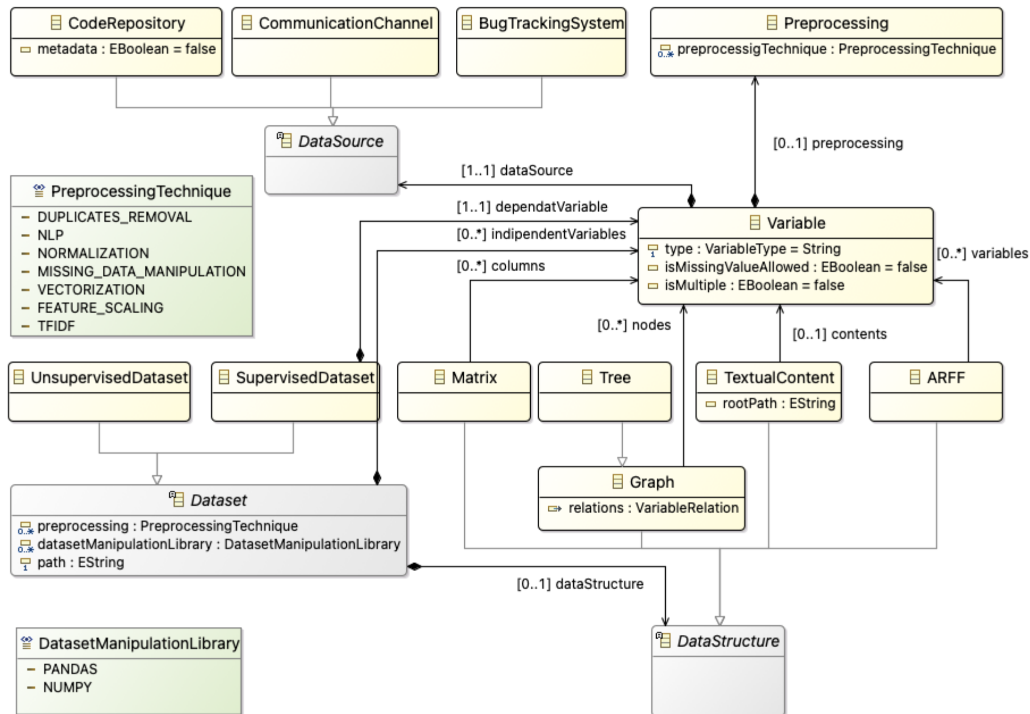


Fig. 8.4 The LEV4REC `Dataset` and `DataStructure` metaclasses.

two main types of `ValidationTechnique` that could be used to assess the selected recommender system, i.e., `AutomatedValidation` and `UserStudy`. Figure 8.6 depicts the key concepts related to the `ValidationTechnique` metaclass. Similar to the algorithm's configuration, the user can set different parameters such as metrics, the type of analysis to be conducted, and the programming library to generate the corresponding code. At its current status, LEV4REC supports the generation of automatic techniques, while user studies are not yet implemented. Nevertheless, we can rely on existing works that come in handy to specify such kind of evaluation [244, 27].

Figure 8.7 depicts the `PresentationLayer` metaclass that is devoted to retrieving produced recommendations from the used IDE e.g., VSCode and Eclipse. `PresentationLayer` relies on a `WebService` that allows two main user interactions, i.e., proactive and reactive. The former continuously monitors the user's context and performs some actions according to certain conditions. Contrariwise, the latter reacts to a specific `UserEvent` that directly triggers the action. Both interactions are managed by `RecommendationHandler` that also rules `RecommendationUsage`, i.e., how the user makes use of the returned items. In this respect, transformative usage acts directly on the retrieved recommendations by modifying them. For instance, if the system suggests code snippets, they can be adapted according to the

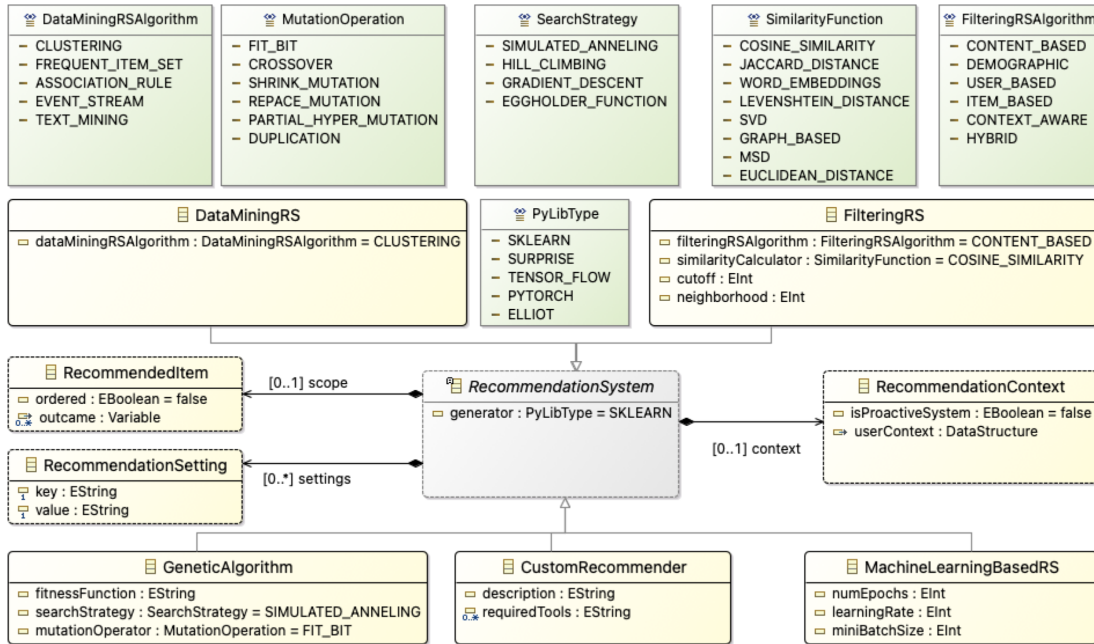


Fig. 8.5 The LEV4REC `RecommenderSystem` metaclass.

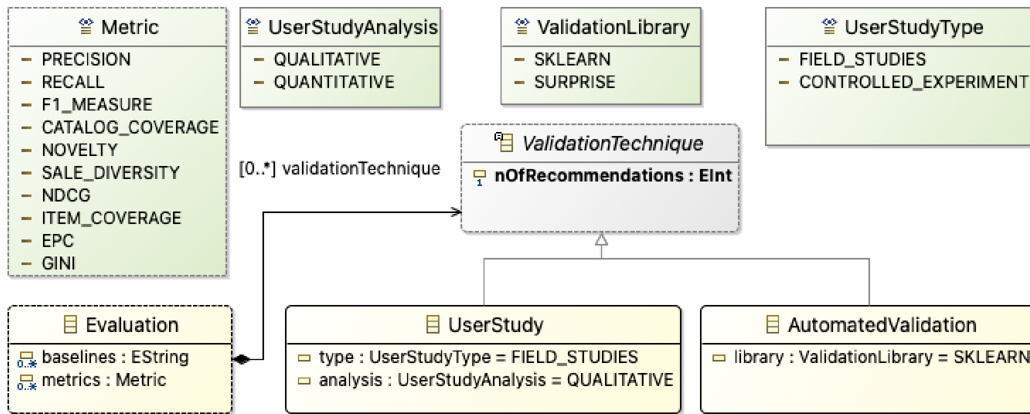


Fig. 8.6 The LEV4REC `ValidationTechnique` metaclass.

actual context. Meanwhile, the user can visualize the delivered items without the possibility of modifying them when the `RecommendationUsageType` instance is set to visualization. According to the type of the selected interaction, the system provides `GUIElement` components capable of handling different behaviors in a specific graphical interface.

8.2.3 RS Code Generation

The final step of the process shown in Fig. 8.1 is the generation of the actual source code for the selected and configured RS components. Starting from an input *RS Configuration*, LEV4REC produces the corresponding source code implementing the designed RS by relying on template engines that have been successfully employed to support this task [20].

The code generation phase has been developed by using Acceleo.[12] It is a well-known technology in model-driven engineering, and it supports the development of code generators in terms of dedicated templates.[13] Acceleo templates identify repetitive and static parts of the system being generated and embody specific queries on the source models to fill the dynamic elements.The rationale behind the selection of Acceleo is twofold: *(i)* it provides a well-defined syntax to specify the templates, and *(ii)* such a generation technique can deliver output in different programming languages apart from Python.

Listing 8.1 An explanatory Acceleo template.

```
[template public generateFilteringRS (algo : FilteringRS )]
[ if (algo. filteringRSAlgorithm = FilteringRSAlgorithm :: USER_BASED)]
is_user_based=True
[/ if ]
[ if (algo. filteringRSAlgorithm = FilteringRSAlgorithm :: ITEM_BASED)]
is_user_based=False
[/ if ]
neighborhood=[algo.neighborhood/]
[ if (algo. similarityCalculator = SimilarityFunction ::
COSINE_SIMILARITY)]
sim_funct='cosine'
[/ if ]
[ if (algo. similarityCalculator = SimilarityFunction :: MSD)]
sim_funct='msd'
[/ if ]
sim_settings = {'name': sim_funct, 'user_based': is_user_based }
algo = KNNWithMeans(k=neighborhood, sim_options=sim_settings)
[/ template ]
```

To support the generation of a recommender system, we devise a hierarchical structure in which we provide different templates for each metaclass described in Section 8.2.2. In such a way, the system can generate a custom instance of each component that can be even fine-tuned to experiment with different configurations. As an example, Listing 8.1 depicts the Acceleo template to generate the source code to adopt a recommendation algorithm based on the collaborative filtering technique. The template uses the key element specified in the model to generate source code. In particular, this component automates the setting of parameters and the evaluation of results. In particular, the *algo* entity represents an instance of the

[12] https://www.eclipse.org/acceleo/
[13] https://wiki.eclipse.org/Acceleo/User_Guide#Templates

`FilteringRS` metaclass in Fig.8.5. According to the features selected by the RS developer, this component automates the setting of the parameters to produce the corresponding source code. For instance, the *is_user_based* variable can be *True* or *False* depending on the chosen filtering technique, i.e., either user-based or item-based. Similarly, the RS developer can choose a different similarity function by setting the corresponding attribute in the LEV4REC model.

8.2.4 Deploying LEV4REC

8.2.4.1 IDE Integration

As depicted in Fig. 8.3, the `PresentationLayer` concept is specialized in different sub-concepts that define the way where the recommendations are provided to the users. For example, by using `WebServices` instances, developers dictate that the recommender system should be deployed as a Web service providing recommendations by means of a REST API.[14] In our implementation, we generate Web services in Flask, a popular Python framework used for developing Web applications.

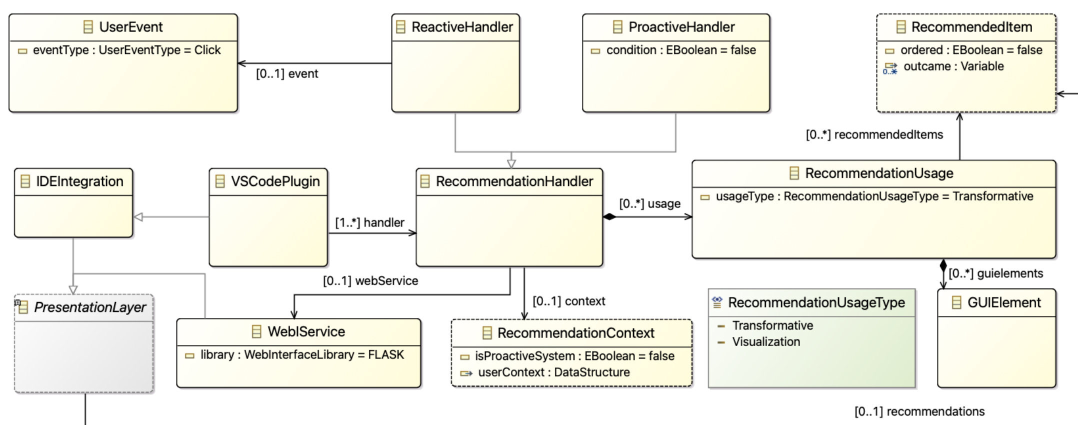


Fig. 8.7 The LEV4REC `PresentationLayer` metaclass.

An example of how generated services can be used to integrate the recommender system into different IDEs is shown in Fig. 8.8. In particular, the left part shows the Swagger UI interface generated from the OpenAPI specification[15] of the developed and deployed RS, while the right part of Fig. 8.8 depicts two IDE integrations, i.e., Eclipse and VS Code, that query the services to provide recommendations to the users.

[14] https://datatracker.ietf.org/doc/html/rfc7231
[15] https://swagger.io/specification/

Swagger UI generated from OpenAPI specification

Eclipse integration

VS Code integration

Fig. 8.8 Using LEV4REC RSs from IDEs.

8.2.4.2 Web-based editor

The architecture of the web-based editor of LEV4REC is represented in Figure 8.9. The whole infrastructure is composed of a REST controller① developed using Spring boot framework equipped with a gateway to handle the different requests and a domain-specific editor ② supporting the fine-tuning of the system exploiting Eclipse modeling framework (EMF) utilities. The user can eventually generate either the system specification ③ in Python plain text or deploy a web API written using the Flask framework. We detail each phase of the proposed tool in the following subsections.

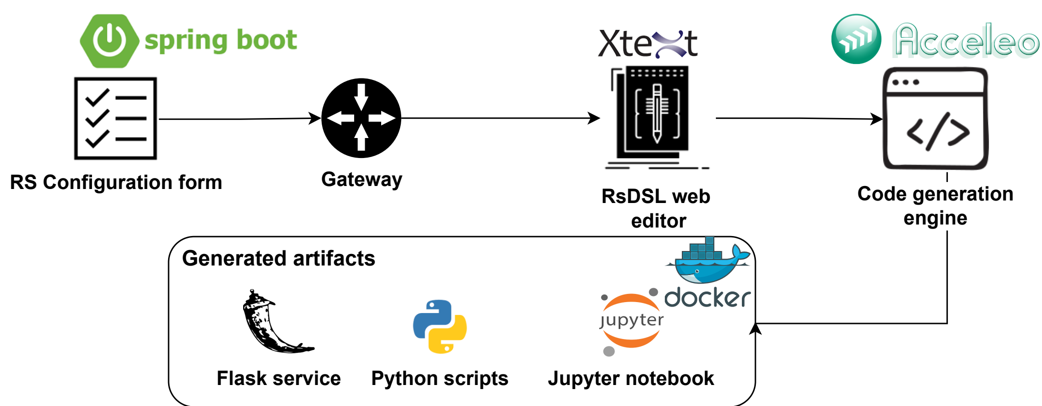


Fig. 8.9 The LEV4REC web-based editor architecture.

Spring REST controller To enable the selection of the needed features, we implement a configuration form that allows the user to choose the high-level components of the wanted RS, e.g., the dataset type, the recommendation algorithm, to list a few. In our previous work, we employ the FeatureIDE plugin [256], an Eclipse-based technology to define a feature model [255, 82] enabling in such a way the selection of crucial components of the system. Furthermore, this kind of model is equipped with several constraints among the different components to drive the user at the design time. Even though the FeatureIDE tool supports the activity properly, we cannot integrate it into a web-based platform. Therefore, we develop a Spring boot project using the Spring initializer API[16] by selecting the standard Maven libraries to build a REST controller. The rationale behind this choice is that *i)* Spring speeds up the development of a web application written in Java by offering several functionalities and *ii)* it is capable of handling a data model object that allows the manipulation of real-world entities. In the scope of our work, we rely on the Spring data model to resemble the functionalities of the FeatureIDE plugin by developing a dynamic web form filled with the

[16] https://start.spring.io/

elicited RS features depicted in Figure 8.10. Using the RS configuration form, the user can easily set up a valid configuration that is sent to the next component, i.e., the DSL web editor. To fill dynamically the form, we employ the Thymeleaf template engine,[17] a well-founded technology that supports the integration with the Spring framework.

LEV4REC Feature model

Quickly and easily set your recommender system preferences.

Algorithm Component

☐ **Collaborative filtering**
☐ **Content based**
☐ **Classification**
☐ MiningAlgorithm

Dataset component

Data structure

☐ TextualData
☐ GraphData
☐ Matrix
☐ ARFF
☐ Tree

Dataset type

☐ UnsupervisedDataset
☐ SupervisedDataset

Fig. 8.10 A fragment of the RS configuration form.

Since the data model is a static object, the constraints among the features have been implemented by using a Javascript validator automatically generated using the LEV4REC code generator component. The implemented constraints act on the high-level components as well as on their subcomponents. For instance, if a Supervised Dataset feature is selected, it is not possible to choose any Unsupervised Algorithm. Meanwhile, some features can be selected independently from the others, e.g., standard preprocessing techniques that can be applied to any dataset. In the original implementation, the validation of such rules is handled by the FeatureIDE Ecplise plugin using a dedicated configuration XML file. Replicating all FeatureIDE functionalities goes beyond the scope of the presented work and we see a complete implementation as possible future work. Once the form has passed the abovementioned formal check, the system sends the data selected to the DSL editor. To this end, we equip the developed Spring architecture with a gateway developed with Spring

[17] https://www.thymeleaf.org/

Cloud component[18] that is able to remap each request to a different path. Due to the nature of the developed editor, requests from external sources are not allowed. Thus, LEV4REC makes use of the gateway to overcome this issue by remapping each component to a proper request.

EMF editor and generation Once the component configuration has been defined, the user can enrich the elicited components with the DSL editor deployed by using the Xtext framework. As discussed in our prior work [71], LEV4REC relies on a metamodel composed of different abstract concepts needed to generalize any recommender systems. Moving towards a more user-friendly platform, a custom grammar has been generated starting from the original metamodel using Xtext utilities. In such a way, users who are not familiar with meta-modeling can directly employ the web editor equipped with all needed features. In particular, the system makes use of *RsDsl.xtext* file to generate the whole grammar by exploiting EMF utilities, i.e., ecore concepts and commons terminal. Afterward, the user can enrich all these elements by selecting additional attributes, e.g., the data source file, algorithm parameters, the number of evaluation steps, to name a few. The generated DSL web editor is depicted in Figure 8.11.

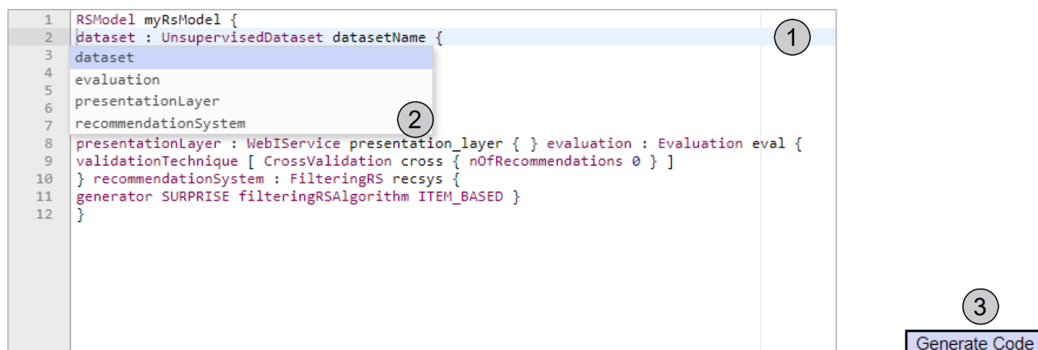


Fig. 8.11 The DSL web editor.

To facilitate a proper specification, the user can exploit the editor features, i.e., syntax highlighting and autocompletion. In such a way, LEV4REC drives the user during the whole design process by providing insightful hints. For instance, if a *FilteringRs* algorithm has been specified, the editor can suggest additional parameters specifically designed for this type of strategy, e.g., neighborhood size or the similarity function, by simply selecting them. Furthermore, we equipped LEV4REC with a tailored Xtend formatter[19] to assist the user

[18] https://spring.io/projects/spring-cloud
[19] https://www.eclipse.org/Xtext/documentation/303_runtime_concepts.html

during the editing operations, thus improving the readability of the written specification. Concerning the interal web-based architecture, the internal Xtext engine generates all the needed components to run the web application, i.e., the webserver and the servlets with the deployed services. In particular, Xtext makes use of Jetty,[20] a well-founded web container that is easily embeddable in several frameworks. The next step involves the generation of the actual code using the abovementioned specification using the Acceleo framework that offers a set of dedicated templates [21]. Such modules identify repetitive and static parts of the system being generated and embody specific queries on the source models to fill dynamic elements. As stated in Section 8.2, an Acceleo generator needs a template and a model that conforms to a metamodel. Since the web editor is built on top of a derived DSL that avoids the usage of an EMF model, LEV4REC has a dedicated component in the Spring architecture that serializes the content edited by the user in a suitable model, thus enabling the generation phase. Listing 8.1 the main Acceleo template employed to generate all the specified components. It is worth noting that a hierarchical structure has been devised to cover the generation of all elicited components, including both the configuration form and DSL editor entities. In other words, the LEV4REC code generator embodies the user design choices from the initial specification to the fine-tuning phase.

8.3 Use cases

By following the process presented in the previous section, we make use of LEV4REC to design, tune, and deploy two existing recommender systems, i.e., a k-nearest neighbour-based algorithm (named KNN hereafter) [250], and AURORA [176]. KNN aims to address the scalability problem in personalized recommendations; it combines an adaptive version of the KNN (AKNN) and ontologies to recommend relevant items for any user. In the scope of the work, we elicit this approach since it provides movie recommendations, which is a well-known application domain [74]. Furthermore, we implement KNN by relying on the Surprise library.[22] Meanwhile, AURORA classifies metamodels using a feed-forward neural network model trained with a curated labeled dataset. To feed the underpinning model, feature vectors are extracted using three encodings, i.e., uni-grams, bi-grams, and n-grams. Like the KNN methods, we resemble the AURORA key functionalities by relying on an external Python library, namely Scikit-learn,[23] which provides all the needed building blocks.

[20] https://www.eclipse.org/jetty/
[21] https://wiki.eclipse.org/Acceleo/User_Guide#Templates
[22] http://surpriselib.com/
[23] https://scikit-learn.org/

We discuss the development of KNN and AURORA in the following subsections. However, as we mentioned in Section 1.1, the proposed platform does not support the generation of a dataset at the current stage of development. Thus, in the scope of this evaluation, we focus on the specification, configuration, and generation of the two RSs.

8.3.0.1 RS Feature Selections

The first step involves the feature selections of the two systems by employing the devised feature model. Figure 8.12 shows a fragment of the KNN and AURORA specifications, which have been done utilizing LEV4REC. To resemble the KNN peculiarities, a *Supervised dataset* and *user-based collaborative filtering* algorithm are needed. To this end, the fragment of the selected features depicted in Fig. 8.12a shows that the *UserBased* feature has been selected accordingly.

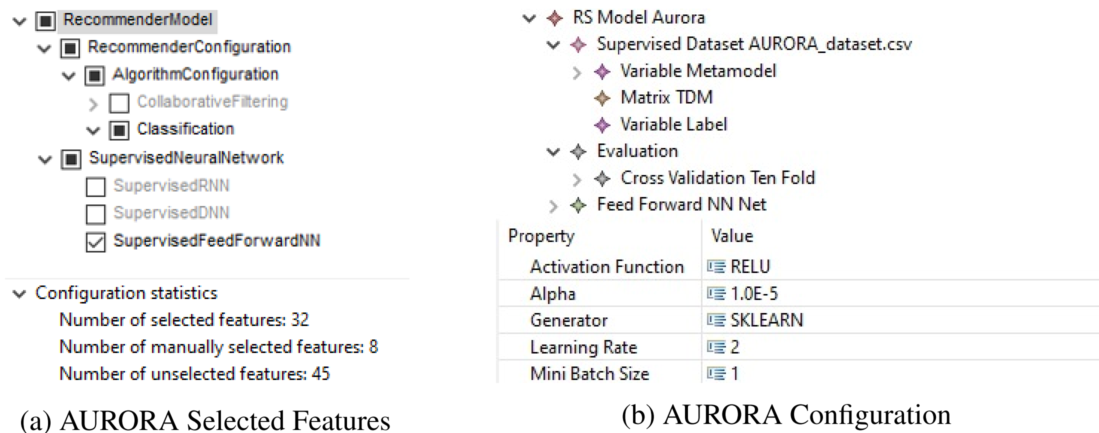
It is worth noting that defining a partial set of features is sufficient to resemble the proposed environment, which relies on well-defined constraints to obtain a valid configuration. For instance, as shown in the lower side of Fig. 8.12a, the user manually selected eight features, and the environment automatically disabled 47 features and selected the remaining 24 ones.

(a) KNN Selected Features

(b) KNN Configuration

Fig. 8.12 Small fragment of the KNN specification.

Similarly, feature selections for AURORA can be easily specified as in Fig. 8.13a. Since the tool exploits a *supervised classifier*, the RS developer is forced to use once again a *supervised dataset*. It is worth noting that such a constraint is automatically set using the feature model. Thus, the RS developer can elicit the wanted component without taking care of possible violations. Concerning the underpinning algorithm, we defined a *FeedForward NN* concept to resemble the original behaviour of AURORA.



(a) AURORA Selected Features

(b) AURORA Configuration

Fig. 8.13 Small fragment of the AURORA specification.

8.3.0.2 RS configurations

Once the features of KNN and AURORA have been selected, as shown in the previous section, an initial version of their configuration models is automatically generated by LEV4REC. We refined such models to provide missing parameters. Concerning the KNN configuration, it includes the specification of a *Variable* element that represents the data sources exploited to perform the recommendations, i.e., the users and the rated movies.

After the dataset definition, the developer can configure the collaborative filtering algorithm by setting different parameters. The user can eventually set an *Evaluation Strategy* to assess the resulting system in terms of available *Metrics*. Moreover, as shown in the model fragment depicted in Fig. 8.12b, we have specified the number of folds and number of wanted recommendations for the selected cross-validation method.

Concerning AURORA, the employed configuration is shown in Fig. 8.13b. As we can see, the *Variable* concept has been used to describe *Metamodel*s since AURORA was originally conceived to categorize them given an initial *Label*. In such a way, we can reuse the same notation to describe different algorithms and input data. Afterwards, according to the available hyperparameters, the RSD can customize the feed-forward neural network selected in the previous phase, i.e., the RS feature selection. For instance, a possible fine-tuning is depicted in the lower part of Fig. 8.13b where the *learning rate* has been set to 2 and *mini batch size* to 1. Furthermore, since LEV4REC targets the Sklearn library when generating the actual code, the user can also set an activation function chosen among the available ones, i.e., *identity*, *logistic*, *tanh*, and *relu*.

Listing 8.2 Excerpt of the generated code for KNN.

```
# DATASET
import pandas as pd
from surprise import Dataset
data = Dataset.load_builtin('ml-100k')
# ALGORITHM SETTINGS
is_user_based=False
neighborhood=40
cutoff=5
sim_funct='cosine'
sim_settings = {'name': sim_funct,
'user_based': is_user_based}
from surprise import KNNWithMeans
algo = KNNWithMeans(k=neighborhood, sim_options=sim_settings)
# EVALUATION SETTINGS
from surprise.model_selection import KFold
from collections import defaultdict
threshold = 3.5
k=10
n_splits=10
kf = KFold(n_splits=n_splits)
for trainset, testset in kf.split(data):
algo.fit(trainset)
predictions = algo.test(testset)

user_est_true = defaultdict(list)
for uid, _, true_r, est, _ in predictions:
user_est_true[uid].append((est, true_r))
precisions = dict()
precisions[uid] = n_rel_and_rec_k / n_rec_k if n_rec_k != 0 else 0
recalls[uid] = n_rel_and_rec_k / n_rel if n_rel != 0 else 0

precision= sum(prec for prec in precisions.values()) / len(precisions)
recall =sum(rec for rec in recalls.values()) / len(recalls)

f1_measure=(2*precision* recall) / (recall + precision)
```

Listing 8.3 Excerpt of the generated code for AURORA.

```
# DATASET
dataset=pd.read_csv('AURORA_train.csv')
X = dataset.iloc[:,:-1].values
y = dataset.iloc[:, -1].values
#PREPROCESSING
from sklearn.feature_extraction.text import CountVectorizer
sc = CountVectorizer(ngram_range=(1,1))
# ALGORITHM SETTINGS
from sklearn.neural_network import MLPClassifier
solver='adam'
alpha=1e-5
hidden_layers=(5, 2)
random_state=1
clf = MLPClassifier(solver=solver, alpha=alpha,hidden_layer_sizes=
    hidden_layers, random_state=random_state)
# EVALUATION SETTINGS
n_splits=10
...
from sklearn.model_selection import KFold
kf = KFold(n_splits = n_splits)
for train, test in kf.split(X):
X_split, X_test, y_split, y_test = X[train],X[test], y[train], y[test]
...
X_train = sc.fit_transform(list_train)
X_test = sc.transform(list_test)
clf.fit(X_train, y_split)
y_pred = clf.predict(X_test)
from sklearn.metrics import precision_score
precision = precision_score(y_pred, y_test, average=None)
prec_all = prec_all + sum(precision)/len(precision)
from sklearn.metrics import f1_score
f1 = f1_score(y_pred, y_test, average=None)
f1_all = f1_all + sum(f1)/len(f1)
```

8.3.0.3 Generation of RS source code

Once the configuration model has been finalized, LEV4REC can generate the source code of the modeled RS. The platform exploits Acceleo templates to produce the Python code as described in Section 8.2.3. This section explains how LEV4REC generates code for the two considered systems. As previously mentioned, currently, we support two different libraries to cover the two use cases, i.e., Scikit-Learn and Surprise. Furthermore, our tool

can be conveniently extended to work with other Machine Learning frameworks, such as TensorFlow or Keras.

The LEV4REC code generator produces the needed Python scripts by adhering to a predefined structure. First, the dataset loading relies on the *pandas* library. Then, LEV4REC configures the algorithm parameters according to specified values in the design phase. Finally, the generated RS evaluates the algorithm by computing the metrics previously selected by the designer.

Listing 8.2 and Listing 8.3 represent fragments of the generated code for KNN and AURORA, respectively. Concerning the former, we exploit the Surprise library as it supports a set of well-defined collaborative algorithms. We choose the K-NN mean algorithm that resembles the original CrossRec settings. The resulting source code is then used to run experiments on a real-world dataset. Afterwards, the evaluation phase is conducted to study the results using common quality metrics in Information Retrieval [155], i.e., precision, recall, and F_1 score.[24] Similar to KNN, in the generated code for AURORA in Listing 8.3, we load a predefined dataset to test the capability to resemble the system's structure. The underlying feed-forward neural network is implemented by exploiting the Scikit-learn MLP class that provides similar features. Finally, the performance is assessed using the cross-validation technique to compute the abovementioned metrics.

The presented code is inserted in a Jupyter Notebook, a well-known open-source format that can be executed directly on several platforms, e.g., JupyterLab,[25] or Google Colaboratory.[26] In such a way, LEV4REC provides an agnostic representation that allows for flexibility in choosing the executing environment.

8.4 Evaluation strategies

To demonstrate the applicability of our approach, this section presents an empirical evaluation consisting of the reimplementation of different recommender systems employing LEV4REC and comparing the obtained results with their original implementations.

We define the research question in Section 8.4.1 while Section 8.4.2 presents the datasets employed in the evaluation. The metrics computed are presented in Section 8.4.3, and the automated evaluation process is presented in Section 8.4.4. By employing the focus group methodology, we assessed the qualitative aspects of the LEV4REC as discussed in Section 8.4.5.

[24] For the sake of presentation, we omitted mathematical details in the metrics computation.
[25] https://jupyterlab.readthedocs.io/en/stable/
[26] https://colab.research.google.com/notebooks/intro.ipynb

8.4.1 Research questions

We study the performance of LEV4REC by answering the following research questions:

RQ$_1$: *Is* LEV4REC *capable of resembling the key functionalities of existing recommender systems?* In this research question, we assess the capability of LEV4REC in resembling the key functionalities of two existing recommender systems that support software development, i.e., KNN and AURORA. To this end, we employ LEV4REC to generate these systems from the design to the actual deployment. Afterward, we validate the obtained results by using a set of well-known metrics.

RQ$_2$: *Is* LEV4REC *useful to support the development of recommender systems?* LEV4REC has been conceived as an agnostic and extensible framework to support different types of RSs. This research question aims to demonstrate to what extend the tool is useful to support the development of recommender systems.

8.4.2 Datasets

To evaluate each system, we elicit two different datasets belonging to two different domains, i.e., movies, and MDE. The KNN algorithm has been validated on MovieLens dataset [105], a well-known *supervised dataset* widely used in the recommendation system domain to test collaborative filtering approaches. The data was collected through the MovieLens website[27] and included 100,000 ratings from 943 users on 1,682 movies plus several demographic user information, e.g., age, gender, and occupation. To feed the underpinning KNN algorithm, the data is encoded as a user-item matrix where each row is represented as follows:

```
user id | item id | rating
```

where the user id represents the user, the item id the rated movie, and the rating is the corresponding vote expressed from 1 to 5. We used the same dataset [300] that has been considered to evaluate AURORA while testing the LEV4REC's performance. The dataset comprises 555 metamodels with 9 different application domains, i.e., Bibliography, Conference management, Bug/issue tracker, Build systems, MS Office products, Requirement/use case, Database, State machines, and Petri nets.

Table 8.2 summarizes the features of the examined dataset as well as the corresponding tools. As we can notice, the input data has different dimensions and format, i.e., AURORA

[27] https://grouplens.org/datasets/movielens/100k/

Table 8.2 Overview of the examined datasets.

System	Dataset	Data type	Features
KNN	Movielens-100k	Matrix	1,682 movies rated by 943 users
AURORA	Ecore dataset	Textual Data	555 metamodels labeled with 9 categories

has to be fed with textual data while a matrix format is enough for the KNN-based approach. Such a heterogeneous knowledge can be handled by LEV4REC by using the feature model and the high-level concepts as discussed in Section 8.2.

8.4.3 Metrics

To evaluate the generated systems, we consider precision, recall, and F1 score that are widely used in the information retrieval domain [107]. First, we define the following notations:[28]

- *True positive (TP)*: is the outcome where the system recommends the proper item;
- *False Positive (FP)*: is the outcome where the system has suggested the wrong item;
- *False Negative (FN)*: is the outcome where the system doesn't recommend an item that should be included in the delivered list.

Precision: This metric evaluates the ratio of number of correctly predicted items to the total number of retrieved items

$$P = \frac{TP}{TP+FP} \tag{8.1}$$

Recall: It measures the impact of the false positive on the recommended items.

$$R = \frac{TP}{TP+FN} \tag{8.2}$$

F_1 score: It represents the harmonic mean of the two previous metrics.

$$F_1\ score = \frac{2 \times P \times R}{P+R} \tag{8.3}$$

8.4.4 Automated evaluation

The evaluation process conducted for the considered systems is depicted in Fig. 8.14. We adopt the ten-fold cross-validation technique consisting of splitting the *Initial data* of each tool into two different dataset, i.e., *train* and *test* data. The former is used to feed the examined

[28] This notation holds for all the considered systems in the evaluation. Thus, the term *item* refers to a movie or a predicted label if KNN or AURORA is considered, respectively

recommender system implemented using LEV4REC. It is worth mentioning that each tool has been trained with data of different natures. For instance, the KNN algorithm needs a matrix composed of users, movies, and the corresponding rate. Meanwhile, AURORA is fed with a set of labeled metamodels employing NLP encoding.

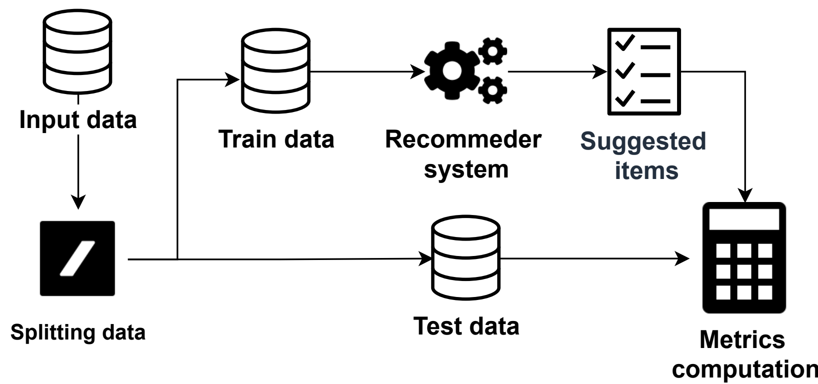


Fig. 8.14 The evaluation process.

The *test* data is used to validate the two systems. In particular, we used 80% of the initial data as training in the conducted study and the left part as testing. Afterwards, the *Metrics computation* process evaluates the performances of the considered recommender system by computing the metrics described in Section 8.4, i.e., precision, recall, and F1 score. To this end, the *suggested item* returned by the system are compared with the elicited test data. It is worth noting that this process takes place for each evaluation round.

8.4.5 Focus group evaluation

To assess the qualitative aspects of LEV4REC, we conducted a focus group study, a well-founded methodology in software engineering [133]. This technique is described as a carefully planned discussion designed to obtain feedback from a group of experts in a specific domain. Though the method is not suitable for quantitatively evaluating a given approach [78], it can provide a fast and cost-effective means to obtain qualitative feedback on the proposed methodology. Furthermore, this methodology has been recently adopted to evaluate a tool that supports the modeling of complex software systems [146]. In the scope of this work, we organized a focus group to assess the capability of LEV4REC by discussing the provided critical features, i.e., the level of automation, the usability of the user interface, the completeness, and extensibility.

Concerning the participants, the focus group involved four academic experts recruited from different universities plus one participant from the industry. The selection process considered their expertise in the relevant fields of study, i.e., recommender systems, low-code

engineering, software product lines, and model-driven engineering. One of the academic participants devises several algorithms in the recommender system domain, while the industrial participant is involved in developing low-code platforms. The rest of the members come from the modeling area. Finally, one author of the paper took the moderator role to drive the discussion.

Table 8.3 Participants to the focus group and their expertise.

Participant	Recommender Systems	Low-code Engineering	Software Product Lines	Model-driven Engineering
P1	Expert	Outsider	Outsider	Outsider
P2	Good Knowledge	Good Knowledge	Good Knowledge	Good Knowledge
P3	Expert	Expert	Expert	Expert
P4	Familiar	Good Knowledge	Familiar	Expert
P5	Expert	Expert	Expert	Expert

Table 8.3 summarizes the level of expertise self-declared from the anonymized participants for each research subject, using the following taxonomy:

- `Outsider`: the participant declares that s/he is not an expert on the specific research subject;
- `Familiar`: the participant knows the fundamental aspect of the topic even though s/he misses the relevant expertise;
- `Good knowledge`: the participant works actively in the field but s/he may not be updated on the latest literature;
- `Expert`: the participant possesses a deep knowledge of the subject and s/he substantially contributes to the state-of-the-art research.

The discussion has been divided into two parts. In the first part, we introduced principles and notions of recommender systems and model-driven engineering to give a common background. Notable approaches and open challenges related to developing new recommender systems have also been discussed. In the second part, we present LEV4REC and its main features using a pre-recorded demo in which the tool's capabilities were shown. The participants eventually discuss the strengths and limitations of our approach by relying on their perceptions. To conduct the focus group, we identified a list of *discussion questions* (DQs) to highlight better the contribution of our approach related to specific aspects. Concerning the first part, we investigated the following DQs:

- $DQ_{1.1}$: What is your experience with recommender systems in general?

- $DQ_{1.2}$: What is your experience with tools that support the design of recommender systems?
- $DQ_{1.3}$: What is your experience in modeling complex software systems?

Meanwhile, the DQs presented in the second part are the following ones:

- $DQ_{2.1}$: Please comment on the usefulness of LEV4REC in guiding the design of a custom recommender system.
- $DQ_{2.2}$: Please comment on how LEV4REC satisfies the following aspects:
  - $DQ_{2.2.1}$: Automation level in specifying the whole system.
  - $DQ_{2.2.2}$: Usability of the LEV4REC interface.
  - $DQ_{2.2.3}$: Capability of resembling the two presented scenarios.
  - $DQ_{2.2.4}$: Extensibility of the whole tool.
- $DQ_{2.3}$ How could the approach be improved or extended?
- $DQ_{2.4}$ How difficult would it be for a newcomer to use LEV4REC for specifying a recommender system?

8.5 Results

RQ_1: *Is* LEV4REC *capable of resembling the key functionalities of existing recommender systems?* To answer the research question, we run the two recommender systems implemented using LEV4REC on the datasets presented in Section 8.2. Figure8.15 reports the experimental results by running KNN on the MovieLens dataset. It is worth mentioning that the KNN performance resembles the results obtained with the standard version of the algorithm, i.e., precision, recall, and F1 scores are ≈0.60 with the better configuration. However, it is important to remark that our approach cannot resemble the augmented version of the KNN since it relies on a tailored algorithm at the current state of development. Thus, we focus on deploying the standard version of the KNN, which was considered as the baseline in the mentioned work [250].

To highlight the LEV4REC's capabilities in fine-tuning the chosen algorithms, we compare the results obtained with user-based and item-based collaborative filtering. The item-based collaborative filtering technique can improve prediction than the user-based one for all considered metrics. For example, as depicted in Fig. 8.15, the precision distribution is

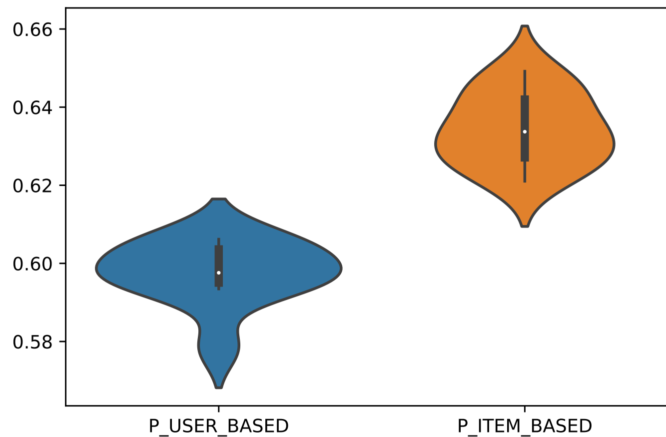


Fig. 8.15 Precision of the KNN-based RS developed with LEV4REC.

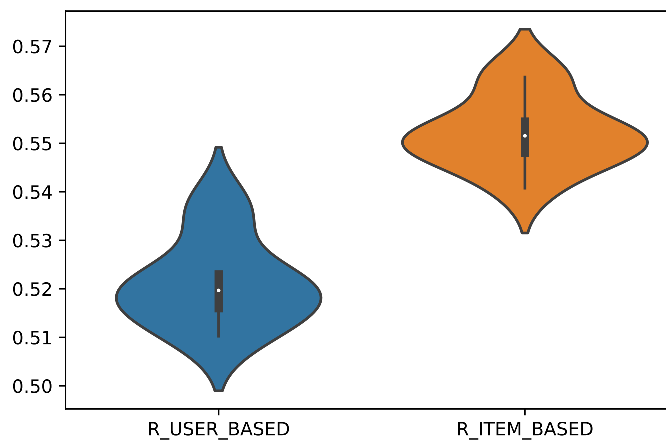


Fig. 8.16 Recall of the KNN-based RS developed with LEV4REC.

centered on 0.64 with the item-based algorithm while the user-based median is 0.59, meaning that it obtains the worst results on average. A similar trend can be observed for the recall and F1 metrics which achieve 0.56 and 0.59 using the item-based technique, as depicted in Fig. 8.16 and Fig. 8.17, respectively. Meanwhile, the user-based median values for the recall and F1 are lower, i.e., 0.52 and 0.56. Altogether, the item-based strategy achieves better results, and this is consistent with the findings of existing work [125, 63]. Thus, we conclude that the produced implementation for KNN can provide recommendations as expected.

The results achieved by the reimplementation of AURORA done with the proposed LEV4REC approach are shown in Fig. 8.18, Fig. 8.19, and Fig. 8.20. It is worth noting

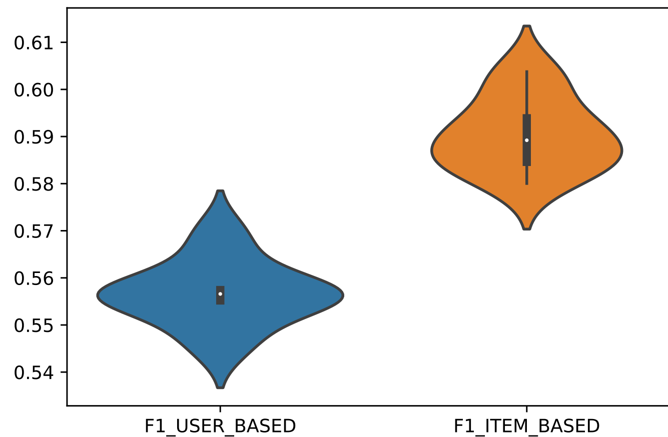


Fig. 8.17 F_1 score of the KNN-based RS developed with LEV4REC.

that LEV4REC succeeded in resembling the AURORA underpinning algorithm used in the classification task: the values of precision, recall, and F1 score are similar to those reported in the original work [176]. Furthermore, according to the work presenting AURORA, the best configuration is reached using uni-gram encoding, which has been resembled by LEV4REC by using the `Count Vectorizer` class provided by Sklearn. In particular, our findings confirm that the proposed environment can mimic AURORA's functionalities, i.e., the original tool achieves better performance when the uni-gram encoding is used. Furthermore, concerning the examined metrics, the results are very high since we are considering a well-curated dataset composed of metamodels that are very similar. Nevertheless, the conducted study aims to evaluate to what extend LEV4REC can conceive the original design of the examined tools rather than improving their performances.

Table 8.4 compares the results obtained by the original tools, i.e., KNN and AURORA, and the corresponding ones that have been developed with LEV4REC. It is worth noting that the proposed approach achieves almost the same values for all the examined metrics on average.

Table 8.4 Comparison with original results.

	Collaborative filtering		**Classification**	
Avg. metrics	KNN	LEV4REC	AURORA	LEV4REC
Precision	0.623	0.634	0.945	0.950
Recall	0.622	0.552	0.938	0.949
F1	0.623	0.590	0.942	0.949

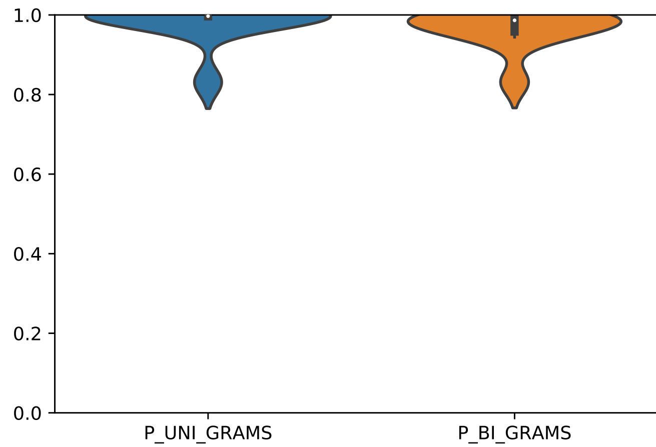


Fig. 8.18 Precision of AURORA as developed with LEV4REC.

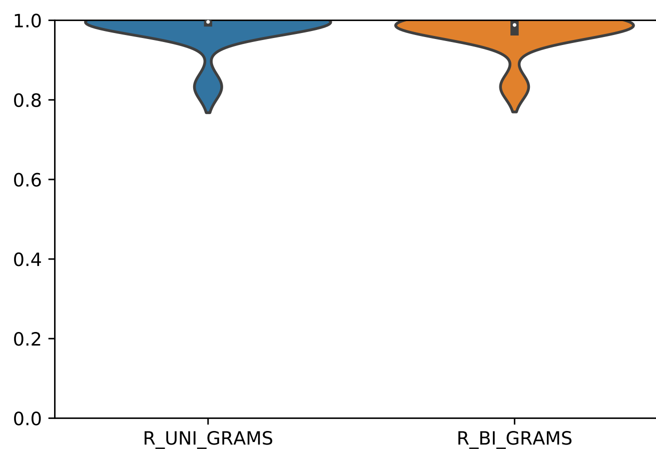


Fig. 8.19 Recall of AURORA as developed with LEV4REC.

Answer to RQ$_1$. The conducted evaluation suggests that LEV4REC can resemble existing recommender systems geared by different algorithms. Furthermore, the obtained results are comparable to those presented in the original studies.

RQ$_2$: *Is* LEV4REC *useful to support the development of recommender systems?* To answer the research question, we conducted a qualitative evaluation following the focus group methodology described in Section 8.4. To facilitate the participation of the heterogeneous group described in Table 8.3, the focus group session was conducted remotely and recorded (with the participants' consent). The session required almost two hours and we transcripted

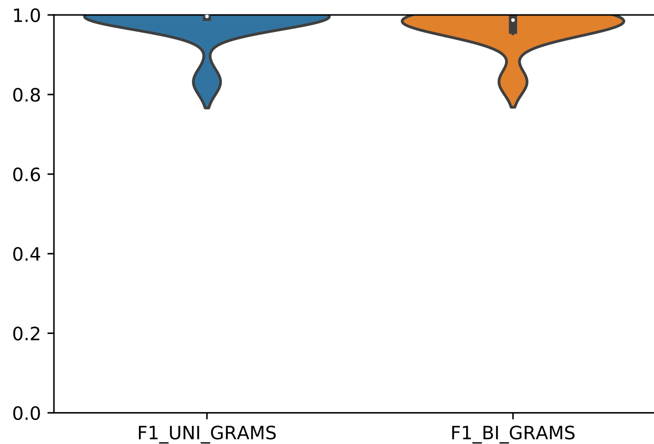


Fig. 8.20 F_1 score of AURORA as developed with LEV4REC.

the participants' answers by anonymizing them.[29] Afterward, we employed the Thematic Analysis Template (TAT) method [47] to examine the results of the focus group session. The rationale behind this choice is two-fold *(i)* the approach is very flexible in analyzing qualitative data; and *(ii)* existing studies in software engineering already make use of it to evaluate focus group results [234, 146]. The definition of the TAT begins by identifying an initial set of themes relevant to the considered context. Alternatively, *a priori* themes can be set according to the author's experience. Then, these elicited concepts can be refined according to further data analysis.

In the scope of our work, we define the themes by relying on the LEV4REC's features that have been investigated using the discussion questions (DQs). Table 8.5 summarize the initial themes, their description, the derived sub-themes, and the final template. Each theme and the corresponding sub-themes are discussed hereafter.

(1) Managing heterogeneity

(1.1) Composition of miscellaneous components

Even though the participants agreed that the design of the recommender systems is ruled by common principles, deploying the actual system could be challenging to the heterogeneity of the data. In this respect, creating a dedicated ontology can facilitate the reuse of existing knowledge. Furthermore, generating a synthetic dataset resembling the users' behaviour can

[29] The anonymized transcription of the meeting is available online: https://github.com/MDEGroup/LEV4REC-Tool/blob/main/focus_group/

Table 8.5 Identified themes and sub-themes using for the TAT methodology.

Nr.	Theme	Discussion	Sub-theme
1	Managing heterogeneity	Each module requires a specific input format to produce intermediary outcomes which are used by the subsequent components	1.1 Composition of miscellaneous components 1.2 Development of source code components
2	Pursuing generalizability	There is the need to generalize the whole design pipeline to support different type of strategies in one platform	2.1 Design and specification of the recommendation process 2.2 Awareness of existing solutions
3	LEV4REC approach	Presenting strengths and weaknesses of LEV4REC in support the design of a generic RSs	3.1 Strengths 3.2 Weaknesses
4	LEV4REC evaluation	Possible evaluation methodology for LEV4REC	4.1 Suitability of the predefined use cases 4.2 Extending the conducted assessment

mitigate the lack of data. Finally, the industrial participant mentioned that the development of recommender systems in industry is tailored to the users' needs, thus involving effort in collecting explicit feedback on each component of the system, e.g., expected outcomes or active context.

(1.2) Development of source code components

According to their experience, the participants claimed that developing source code from scratch requires deep domain knowledge. Therefore, Python is the most used language as a proof of concept to test the feasibility of the system since *(i)* it is easy to learn; and *(ii)* it offers a plethora of external modules that can come in handy in the development of recommender systems. The industrial participant confirmed that Python is adopted in her company to have an initial application prototype. Afterwards, they use this initial specification to implement the final system in different languages, e.g., Java, C++, and C#. Furthermore, all the participants recognized that LEV4REC offers the needed degree of automation to generate the Python scripts, thus simplifying the whole process.

(2) Pursuing generalizability

(2.1) Design and specification of the recommendation process

The industrial participant acknowledged the potential benefits of using the modeling concepts to develop complex systems even though they are currently not used during daily activities. The rest of the academic participants agreed that using the feature modeling combined with the dedicated metamodel is helpful while the user specifies the system.

(2.2) Awareness of existing solutions

Three participants worked actively in developing and evaluating recommender systems by considering different domains, from modeling assistants to multimedia recommenders. In particular, the existing modeling assistant relies on a query mechanism to retrieve valuable modeling artifacts. Concerning the multimedia recommendation, a generic framework that relies on a collaborative filtering technique has been developed by one of the participants. Furthermore, they highlighted that gathering user ratings needed to feed the underpinning techniques was the most difficult part. Finally, the participant with deep knowledge of recommender systems developed a framework to automatically evaluate and test several state-of-the-art recommendation engines, e.g., deep neural networks and collaborative filtering strategies. Altogether, all the participants involved in the actual development agreed that writing the needed code from scratch is daunting and time-consuming. Moreover, the quality of the results is not granted apriori, thus requiring extra time to assess the developed system properly.

(3) LEV4REC approach

(3.1) Strengths

Overall, the participants found that LEV4REC achieves several critical objectives, from the automation of the overall process to the generation of the actual code. Furthermore, the usage of the FOSD methodology in combination with the standard metaprogramming was appreciated by the experts in MDE. Moreover, the participant involved in testing recommender systems remarked that LEV4REC could be used to lower the barriers between the developers and the stakeholders, thus allowing small companies to exploit these complex systems in their daily activities. In this respect, the industrial participant recognized that our tool could support the proof of concept strategy that is currently employed in the company.

(3.2) Weaknesses

During the discussion, some participants expressed some concerns related to two aspects, i.e., the usability of the LEV4REC interface and the extensibility mechanism. Concerning the former, two of the participants suggest providing some default configurations for the state-of-the-art approach. In such a way, newcomers unfamiliar with the Eclipse environment

can easily understand the main capabilities of the modeling environment. On the other hand, concerning the latter, the participant who worked in the modeling assistant domain advised us to introduce well-documented guidelines to describe how to extend LEV4REC. In particular, providing endpoints could allow for integration with external tools and frameworks.

(4) LEV4REC evaluation

(4.1) Suitability of the pre-defined use cases

Although the conducted evaluation included only two use cases, it is suitable to show the generalizability of LEV4REC in supporting the design of a custom recommender system, as almost all participants claimed. Furthermore, one participant mentioned the possibility of generating different variants of the same system to select the best configuration given a set of predefined requirements. In this respect, we see evaluating the results in terms of the identified metrics as possible future work.

(4.2) Extending the conducted assessment

To better discuss the qualitative aspects of the tool, one participant suggested employing the system and study skill methodology to collect feedback from the non-expert user. Another proposed methodology is to define a set of Key Performance Indicators (KPI) that can investigate several aspects of the system, for instance, how the usage of LEV4REC can impact design time. Nonetheless, the definition of KPIs requires a more structured user interface and the implementation of a DevOps architecture where the system is deployed as a stand-alone prototype.

Answer to RQ$_2$. The focus group participants highlighted the benefits of LEV4REC in terms of automation and generalizability, even though the user interface and the extensibility can be enhanced by introducing default configurations and documented endpoints, respectively.

8.6 Threats to validity

This section discusses some threats that may affect the validity of our findings and identifies possible countermeasures to minimize the potential issues.

Threats to *internal validity* concern the selected methodology to generate the RS, i.e., the devised low-code environment. An RS developer who is not familiar with modeling concepts and the Eclipse environment could have some difficulties in using LEV4REC. We provide a constraint-based mechanism to support users while selecting the RS features to mitigate such

an issue. Also, the RS configuration phase is supported by a metamodel, which enforces a well-defined structure to the specifications being defined.

External validity is related to the generalizability of the approach. This work integrated the Surprise and Scikit-learn utilities even though many libraries might be alternatively used. However, LEV4REC can be extended to support different algorithms and programming languages. Finally, selecting just two approaches for the evaluation could be seen as a threat to the generalizability of LEV4REC. However, the two systems correspond to two main algorithms in the recommender system domain, i.e., collaborative filtering technique and feed-forward neural network. Furthermore, we mitigate this threat by conducting a focus group discussion where the five domain experts analyzed the benefits and issues of the approach. Although it is a lightweight evaluation compared to a user study, we followed a rigorous process that has been used in recent works to evaluate software engineering tools.

8.7 Conclusion

Due to the urgent need to support the development of recommender systems, LEV4REC has been proposed as a workable solution to assist developers that do not have strong experience in designing and programming recommender systems. Our approach is a low-code environment to foster an RS's design, configuration, and deployment from scratch using such a cutting-edge paradigm. LEV4REC is flexible and extensible as it relies on three core techniques, i.e., feature model, metamodel, and Acceleo templates. Starting from a feature model, RS designers can specify the system's features and then progressively enrich a configuration model automatically generated out of the selected features. Once the RS configuration has been refined, the system employs a model-driven code generator to produce the actual code of the specified RS. LEV4REC allows developers to refine the produced system by experimenting with different algorithms, experimental settings, and evaluation metrics.

We evaluated the approach empirically by reimplementing two existing RSs that rely on different algorithms, i.e., collaborative filtering technique and feed-forward neural network. To discuss qualitative aspects of the proposed approach, we interviewed five domain experts by employing the focus group methodology, widely used in software engineering, to gather feedback on the benefits and limitations of the proposed approach.

Chapter 9

Adversarial Attacks to Recommender Systems in Software Engineering

To cope with their everyday programming tasks, developers access and browse various information sources [57]. Given the abundance of sources of formal and informal documentation, the problem is not the lack of information, but, instead, its overload [168, 169]. Recently, many studies have been conducted to develop methods and tools–recommender systems for software engineering (RSSE)–to provide developers with automated assistance [101, 203, 182]

The development of RSSE encompasses several phases including the design of the underpinning algorithms or the reuse of existing ones. Machine Learning (ML) techniques are amongst the natural choices that developers take when new recommender systems have to be conceived [220]. This means, for example, that the recommendation of code elements (snippets or APIs) is learned from existing code bases or informal documentation. As a result, the quality of the recommendation depends on the quality of the underlying data, whose noisiness has been previously reported [124].

Even worse, online data used to train recommenders can be exploited for malicious purposes. Adversarial Machine Learning [262, 114] (AML) is a field of study that focuses on security issues in ML systems and, specifically, in recommender systems too [61]. Research has been done to identify probable threats and seek out adequate countermeasures [18, 277]. For example, Anelli *et al.* [18] use shilling attack [241] to manipulate a collaborative-filtering recommender system operated with Linked Data. Also, Wang and Han [277] propose an improvement of the Bayesian personalized ranking technique by exploiting the adversarial training-based mean strategy in collaborative filtering-based recommender systems.

To protect a system against threats, in the first place, it is necessary to be knowledgeable of various types of adversary activities [164]. Such activities generate perturbations to

deceive and disrupt systems by causing a malfunction, compromising their recommendation capabilities.

The ultimate aim of these attacks is to manipulate target items, thus creating either a negative or positive influence on the final recommendations [104], depending on the attacker's intention. For instance, in image classification, an attacker crafts an input image by adding non-random noise in a way that it will be falsely classified by ML models, whilst still being properly recognized by humans [174]. As an example, a panda was recognized as a ribbon by cutting-edge deep neural networks once the input image had been padded with noise, meanwhile humans still correctly perceived the panda from the forged image [95]. AML has been studied in a wide range of domains, e.g., online shopping systems [164] or image classification [174], and addresses both risks and countermeasures. To the best of our knowledge, even though AML is gaining much attention in the software engineering domain, e.g., in testing and applications of DNNs [240], AML has not been investigated in the context of RSSE yet.

This chapter reports two initial investigations that consider two types of RSSEs, i.e., systems to provide third-party libraries (TPLs) and API function calls.

A promising field is enhancing the robustness of ML-based systems. Differently from the profile classification strategy, those approaches aim at improving the underlying algorithm by exploiting feature-space attack models [291]. Even though several approaches propose robust models to support different tasks [260, 115], there is still room for improvements, e.g., handling realizable attacks or moving to generalized robustness.

The first part of the chapter presents an initial study on threats that cause harm or danger to RSSE suggesting third-party libraries and API function calls. The work has been published in the Vision and Emerging results track of the International conference on Evaluation and Assessment in Software Engineering [187]. The candidate contributes to develop the scripts used to simulate the adversarial attacks.

We select these two types of recommendations as they are representative of scenarios in which *(i)* the recommendation is learned from OSS repositories; and *(ii)* the outcome of a malicious recommendation, e.g., the usage of a library or an API, can result in severe security holes [22].

Based on thorough observations and on the analysis of the existing literature, we realized that most of the efforts to improve RSSE have been made to enhance their accuracy. As far as we can see, no work has investigated the problem of using intentionally falsified data to compromise recommender systems' capabilities, as well as conceptualizing countermeasures. In this respect, our work is the first one that brings in the issue of AML in RSSE.

Most RSSE heavily rely on open data sources, such as GitHub, the Maven Central Repository, or Stack Overflow, which can be steered by adversaries [293]. In other words, these systems are vulnerable to attacks equipped with forged input data. Therefore, there is the need for comprehending the likely threats, with the ultimate aim of conceiving counteractions to increase the resilience of RSSE.

We first provide an overview of state-of-the-art API and third-party library recommenders, discussing how they could be potentially affected by AML. Through a literature search from premier venues in Software Engineering for adversarial techniques, we show that there are considerably evident threats to RSSE. Then, we perform a preliminary examination of two existing systems for recommending third-party libraries. The experimental results reveal a worrying outcome: by a simple manipulation, we can seamlessly spoil the final recommendations, putting software clients at risk.

In the second part of the chapter, we present an extended version of our initial investigation which has been thoroughly extended under different dimensions including the analysis of code snippet seeding scenario. This work has been published in the ASE conference [188] and the candidate contributes to the evaluation part by developing the API injector used in the experiments.

First, through a literature analysis on 14 premier venues in software engineering, we show that there has been no work to study the issue of AML in RSSE. Afterwards, we present a qualitative analysis of state-of-the-art API recommenders, underlining their risk of being manipulated by adversarial techniques. Then, we perform an empirical evaluation on three API recommenders using data seeded in real OSS projects. The results reveal a worrisome outcome: by crafting input data to feed the systems, we can manipulate the recommendations, successfully promoting fake/toxic API calls. Last but not least, we devise some possible countermeasures to cope with this type of manipulations.

It is worth noting that the discussed methodologies are limited to these two application domains, thus not considering other types of RSSEs presented in this dissertation, i.e., modeling assistants and OSS tagging systems. While the overall process could be applied also in these domains, it would require a deeper investigation that can be seen as future work.

Outline of the chapter: Section 9.1.1 presents a taxonomy of the attacks and a list of notable RSSEs. The inital investigation on the topic is described in Section 9.1.2 and the preliminary results are discussed in Section 9.1.3. Section 9.2.1 introduces the same issues for API RSSEs. Then, the extended study is presented together with the new research questions in Section 9.2.2 and the corresponding results are reported in Section 9.2.3. We discuss

limitations and possible countermeasures in Section 9.2.4 and Section 9.2.5 respectively. We conclude the chapter by envisioning possible future work in Section 9.3.

9.1 Adversarial attacks to TPL RSSEs

9.1.1 Motivation and background

Classification of attacks Attacks to recommender systems are classified into two main categories as follows [61]:

- *Poisoning attacks* spoil an ML model by falsifying the input data;
- *Evasion attacks* attempt to avoid being detected by hiding malicious contents, which then will be classified as legitimate by ML models.

With poisoning attacks, there are two possible interventions:

- *Push attacks* favor the targeted items, thus increasing the possibility of being recommended;
- In contrast, *nuke attacks* try to downgrade/defame the targeted items [18, 164], compelling them to disappear from the recommendation list.

In the scope of the presented study, we focus on poisoning attacks as they are easy to conduct, yet effective. The remaining attacks are left to our future work.

Potential risks to RSSE for mining libraries and API calls We review notable RSSE that support the development of software projects by delivering third-party libraries and API calls. Table 9.1 lists the systems according to their functionality in chronological order. By studying their internal design, we discuss possible vulnerabilities according to the previously-given categorization of attacks.

▷ **Library recommendation.** LibRec [259] recommends libraries using a combination of rule mining and a collaborative-filtering technique to mine libraries from projects similar to the one being developed. LibCUP [229] suggests libraries that have strong ties by using a clustering approach to identify and recommend co-usage patterns. LibD [142] provides libraries to Android apps using a clustering technique. First, it decompiles applications to build a control flow graph composed of packages, classes, and methods belonging to the projects. Then, the graph is used to extract features, and grouped by a similarity function.

Table 9.1 Notable RSSE for mining libraries and APIs.

	System	Venue	Year	Data source
Library rec.	LibRec [259]	WCRE	2013	GitHub
	LibCUP [229]	JSS	2017	GitHub
	LibD [142]	ICSE	2017	Android markets
	LibFinder [197]	IST	2018	GitHub
	CrossRec [182]	JSS	2020	GitHub
	LibSeek [106]	TSE	2020	Google Play, GitHub, MVN
API rec.	MAPO [298]	ECOOP	2009	SourceForge
	UP-Miner [276]	MSR	2013	Microsoft Codebase
	DeepAPI [101]	ESEC/FSE	2016	GitHub
	PAM [83]	ESEC/FSE	2016	GitHub
	FINE-GRAPE [233]	EMSE	2017	GitHub
	FOCUS [181, 177]	ICSE	2019	GitHub, MVN

Ouni *et al.* propose LibFinder [197], providing libraries based on a multi-objective search-based algorithm. Being built with a collaborative-filtering technique, CrossRec extracts libraries from similar projects [182]. LibSeek [106] relies on a matrix factorization technique to deliver relevant libraries for mobile apps, obtained by collecting neighborhood information, i.e., characteristics of similar libraries.

As it can be seen, all the considered systems leverage open sources, e.g., GitHub or Android markets, for training. Moreover, they mine libraries using similarity-based measures, either a similarity function, or a clustering technique. Thus, they are exposed to perturbations with malicious content hidden in OSS projects. We therefore conjecture that, by fabricating projects with bogus data, attackers can favor (push attack) or defame (nuke attack) a library [18, 241, 61]. In other words, they can make a good/useful library out of being recommended, or even worse, promote a bad/malicious library to a higher rank in the recommendation list, so that users of the recommender system would unintendedly adopt it.

▷ **API recommendation.** MAPO [298] recommends API patterns by extracting API related information from the developer's context. The resulting data is clustered and ranked according to their similarity with the client code. In this respect, the system can be fooled with malicious code intentionally inserted into similar projects. Wang *et al.* propose UP-Miner [276] exploiting SeqSim and BIDE to mine from source code. Since UP-Miner relies on a similarity measure, it may recommend to developers malicious code embedded in projects disguised as similar. DeepAPI [101] generates relevant API sequences starting from a natural language query. It employs an RNN Encoder-Decoder to encode words in context vectors used to train the model. As the corpus is collected from GitHub projects, a hostile user can easily inject perturbations during the data gathering phase, i.e., feeding the system with interfered projects. PAM [83] has been proposed to extract relevant API patterns from client code by using the structural Expectation-Maximization (EM) algorithm to infer the most probable

items. The mined API patterns are then ranked according to their probability. Push and nuke attacks could easily modify the final ranking obtained by the tool, i.e., operating on terms' occurrences to favor or defame a certain pattern. FINE-GRAPE [233] delivers relevant APIs by relying on the history of the related files. It parses GitHub projects, discovers, and ranks the relevant API calls according to their history, i.e., methods, annotations, and classes from every API version. FINE-GRAPE is prone to manipulations which forge an artificial history of API calls in GitHub projects. FOCUS [181] suggests APIs by encoding projects in a tensor and using a collaborative-filtering technique. Since it works on data mined from similar projects, FOCUS is not immune from poisoning attacks, i.e., an adversary can create fake projects with toxic APIs and pose them as legitimate to trick FOCUS into recommending these calls.

9.1.2 Proof of concept

We report a preliminary investigation into the implication of AML on two existing RSSE. The study has been conducted on two library recommenders, i.e., LibRec [259] and CrossRec [182] (cf. Table 9.1). Such tools are chosen due to the following reasons. LibRec is a well-established tool, considered to be the first library recommender system. CrossRec is a more recent approach, which has been shown to outperform LibRec. Both tools have the replication package available,[1,2] allowing us to fully make use of the available source code and tailor it to the study needs.

9.1.2.1 Problem statement

We consider a scenario in which, when developing new software, programmers rely on a recommender system to find and use in their code third-party libraries that offer the desired features. For instance, the developer is searching for libraries for parsing JSON documents and the system should be able to suggest those that are used by similar projects. We simulate an attacker attempting to inject a phoney library with malicious code (e.g., by referring to the previous example, a malicious JSON parser), let it be lib^*, into the active projects. Given that, under normal circumstances, the systems would never recommend lib^*, since the library has not been invoked anywhere. To this end, we forcefully favor lib^* using push attacks (cf. Section 9.1.1), so that, in the end, the library will be suggested to projects.

While in many cases experienced developers would simply rely on well-known libraries, sometimes a malicious library, which is properly documented (i.e., providing useful features),

[1] We gratefully thank the authors of LibRec for providing us with the source code of the tool (through private communications).

[2] https://github.com/crossminer/CrossRec

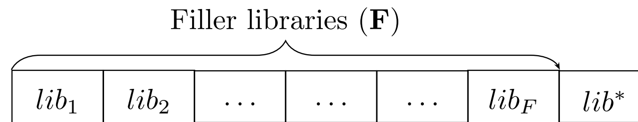


Fig. 9.1 Manipulation of library usages for a fake project.

and (artificially) upvoted, may still be adopted by some. Moreover, while our evaluation is limited to library recommendation, a similar attack scenario could happen for recommenders that suggest code snippets, e.g., MUSE [167], PAM [83], or FOCUS [181, 177].

The name *lib** has been chosen for illustration purposes only, so as to facilitate the reading. In practice, to avoid being detected, an attacker probably gives the library a copycat name, i.e., one that closely resembles a popular library, aiming to disguise it better. In fact, typosquatting has been reported in recent work [60], and this indeed suggests an evident threat to the adoption of libraries. For instance, a malicious library has been named as "*jeIlyfish*" to deceive developers into believing it "*jellyfish*," the authentic library.[3] In this way, developers would adopt the disguised library without hesitation, once it has been suggested by recommenders.

Summary. Altogether, this is to stress that the risk of being fooled by malicious libraries in practice is present. As such, software developers who use recommender systems may suffer if the issue of dealing with this type of attack is not adequately studied.

9.1.2.2 Study design and dataset

Both LibRec and CrossRec suggest to active projects, i.e., the ones under development, by searching for libraries from the most similar projects in the training data [182].

When LibRec and CrossRec learn from existing projects to provide recommendations, they treat a project's library usage as a vector, whose each entry corresponds to a library: 1 means that the library is included in the project, 0 otherwise [259, 182]. To attack a recommender, there are various approaches to populate ratings for fake user profiles [31, 164]. Unfortunately, they cannot be directly applied to forge projects, since they deal with multi-level ratings, while in library recommendation there are only two *ratings* (0 and 1). Thus, to create a training set for the LibRec and CrossRec tools, we have to devise a new method to produce fake projects as follows. A project's vector is divided into two independent parts, namely *filler libraries* and *target library* (cf. Fig. 9.1). We fill in the former with a set of

[3] https://bit.ly/394uETI

libraries from training data; while the latter is set to lib^*. This aims to achieve two goals: *(i)* making the fake project be similar to active projects; and *(ii)* boosting the presence of lib^*.

Table 9.2 Hit ratio @N for LibRec.

		HR@10					**HR@15**				
		γ=3	**γ=4**	**γ=6**	**γ=8**	**γ=10**	**γ=3**	**γ=4**	**γ=6**	**γ=8**	**γ=10**
$\alpha = 5\%$	**k=5**	—	0.154	0.177	0.242	0.219	—	0.154	0.189	0.252	0.237
	k=10	—	0.198	0.273	0.428	0.410	—	0.198	0.317	0.455	0.442
	k=15	—	0.199	0.307	0.541	0.546	—	0.199	0.368	0.580	0.580
	k=20	—	0.177	0.316	0.608	0.637	—	0.177	0.388	0.658	0.670
$\alpha = 10\%$	**k=5**	—	0.247	0.240	0.242	0.210	—	0.247	0.256	0.251	0.237
	k=10	—	0.380	0.418	0.428	0.320	—	0.380	0.449	0.455	0.442
	k=15	—	0.439	0.505	0.541	0.390	—	0.439	0.554	0.580	0.580
	k=20	—	0.443	0.547	0.608	0.431	—	0.443	0.605	0.657	**0.670**

We come across the following question: "*Which libraries should be chosen as fillers, so that the fake project will be incorporated into the recommendation?*" In fact, recommenders heavily rely on similarity [235], thus we come up with the selection of popular libraries as fillers. Our intuition is that, a fake project with libraries widely used by projects is likely to get attention from the recommendation engine. A list of libraries in the training data is selected and sorted in descending order of the number of caller projects. A certain number of the top popular libraries is then randomly selected.

On the attacker's side, α is the ratio of fake projects to the total number of projects (in %); γ is the number of fillers. For LibRec and CrossRec, k is the number of similar projects and N is the cut-off value for the ranked list of recommendations [259, 182]. We conducted experiments using ten-fold cross-validation on a dataset used in our previous work [182] with 1,200 projects and 13,497 libraries. For a testing project, we randomly split its libraries into two equal parts, the first part is used as a query, while the second one is ground-truth data. To measure the effectiveness of the push attacks, we use *Hit ratio HR@N*, defined as the ratio of projects being recommended with lib^* to the total number of testing projects [18]. To measure the accuracy, we use success rate *SR@N*, i.e., the ratio of projects getting at least a matched library (the size of the intersection of the recommended list with the ground-truth data is larger than 0) to the number of testing projects. We experimented with α=$\{5\%, 10\%\}$ as attack size; k=$\{5, 10, 15, 20\}$; γ=$\{3, 4, 6, 8, 10\}$ as fillers' size. The calibration of these parameters allows us to simulate *stress tests*, aiming to investigate the systems' resilience under different working conditions.

We address research questions to study the performance:

- **RQ$_1$**: *Are LibRec and CrossRec prone to push attacks?* By experimenting the two systems on a real dataset collected from GitHub, we are interested in understanding if the push attacks pose a threat to them.

Table 9.3 Hit ratio @N for CrossRec.

		HR@10					**HR@15**				
		γ=3	**γ=4**	**γ=6**	**γ=8**	**γ=10**	**γ=3**	**γ=4**	**γ=6**	**γ=8**	**γ=10**
α = 5%	**k=5**	0.159	0.158	0.145	0.113	0.103	0.178	0.177	0.163	0.138	0.120
	k=10	0.140	0.141	0.165	0.149	0.140	0.184	0.190	0.201	0.185	0.174
	k=15	0.154	0.163	0.188	0.198	0.189	0.200	0.227	0.235	0.250	0.229
	k=20	0.112	0.135	0.207	0.188	0.194	0.168	0.191	0.268	0.235	0.250
α = 10%	**k=5**	0.356	0.346	0.267	0.235	0.210	0.386	0.373	0.291	0.265	0.248
	k=10	0.440	0.430	0.348	0.347	0.320	0.496	0.478	0.388	0.393	0.357
	k=15	0.427	0.445	0.427	0.407	0.390	0.487	0.497	0.488	0.475	0.438
	k=20	0.455	0.424	0.457	0.454	0.431	0.515	0.525	**0.530**	0.514	0.486

- **RQ$_2$**: *How do α and γ affect the attacks' effectiveness?* Both parameters can be tuned by attackers. Thus, it is also important to see how the parameters impact on the efficacy of the attacks. By varying the parameters, we study the influence on the recommendation outcomes of both systems.

9.1.3 Experimental results

We answer the research questions by referring to Table 9.2 and Table 9.3. LibRec does not work on projects with a small number of libraries, this is why there are no results for γ=3. Instead, we can run CrossRec with any $\gamma > 0$.

RQ$_1$: *Are LibRec and CrossRec prone to push attacks?* According to Table 9.2, we see that regardless of the internal configuration of both systems (i.e., k and N), the push attacks always succeed in injecting lib^* to projects. In general, an attack becomes more effective when a larger value of k is used. We analyze the results obtained by running the tools as follows.

With α=5%, for LibRec, we can see that the system recommends the malicious library at different hit ratios. For instance, with γ=4, k=5 we get HR@10 of 0.154, and the hit ratio gradually increases when we use a larger number of libraries. Given that 10 libraries are used, HR@10 reaches 0.219. Moreover, the hit ratio also increases alongside k, i.e., the number of neighbor projects used for recommendations. Remarkably, when k=20, we get a hit ratio HR@10 of 0.637. Similarly, CrossRec also recommends the malicious library by all the experimental settings.

As we have shown in Section 9.1.1, developers would voluntarily adopt the recommended libraries, especially when the fake library is disguised with a typosquatting name, and the ranked list is short, e.g., N=10 or N=15. The adoption of a phoney library indeed poses a threat to the system under development. Altogether, we conclude that adversarial attacks to

library recommender systems are highly probable, and we should neither underestimate nor neglect them.

Answer to RQ$_1$. Even with a simple fabrication, the resilience of LibRec and CrossRec is considerably compromised. Both systems recommend to developers the malicious library, which can put software clients at risk once being invoked.

RQ$_2$: *How do α and γ affect the attacks' effectiveness?* Adversaries pay attention to α and γ, which are under their control and can be tuned to make attacks more effective. The results in Table 9.2 and Table 9.3 show that by adding more fake projects, one can increase the hit ratio by both systems.

For example, with LibRec, for (α=5%, k=10, γ=6) HR@10 is 0.273; and for (α=10%, k=10, γ=6) the corresponding hit ratio HR@10 is 0.418. Similarly, with CrossRec (cf. Table 9.3), for (α=5%, k=20, γ=6), HR@10 is 0.207, while the hit ratio HR@10 for α=10% reaches a value of 0.457. The maximum hit ratio is obtained, i.e., HR@15 = 0.530 when γ=6 and k=20. For CrossRec, the hit ratio decreases when $\gamma > 6$, this means that an attacker should use a small γ to get a higher hit ratio. For LibRec, the maximum HR@15 is 0.670 and obtained when γ=10.

We ran LibRec and CrossRec in two modes, i.e., with and without fake projects, and realized that there is almost no difference in their success rate. Due to space limit, we report only the differences as average *SR@N* in two modes, which is 2.6% and 2% for LibRec and CrossRec, respectively.

This implies that the inadvertent inclusion of fake projects in the recommendation is difficult to notice, and cannot be detected by simply looking at variations in the accuracy values.

Answer to RQ$_2$. By adding more fake projects (α), attackers can increase the number of infected clients in both systems. A small γ is more dangerous to CrossRec, while a large γ causes more harm to LibRec. The introduction of the malicious library does not greatly impact on the recommendation accuracy, thus being an imperceptible incident.

9.1.3.1 Discussions

This subsection discusses the perspective of an attacker who wants to *compromise the security of a system (marked with ✗)*, and that of an administrator who *defends the system against hostile attempts (marked with ✔)*.

While in the evaluation we performed only push attacks, the same methodology can be applied to conduct nuke attacks, i.e., removing a useful library from the recommendations

(✗). Moreover, we tried with only one malicious library, however the evaluation can easily be extended to a set of libraries.

We considered different settings for LibRec and CrossRec, and this aims at studying the systems' capability in various conditions. In practice, an attacker may not be aware of such design, what matters is how she populates fake projects so that the malicious library will be recommended (✗), regardless of the systems' configurations.

To simplify the evaluation, we generated fake projects at the metadata level, i.e., we assume that all projects have been successfully fetched from GitHub.[4] In practice, it is necessary to plant these projects in GitHub to make them appear as legitimate (✗). In fact, such an incident has been recently reported, where hundreds of GitHub repositories promoting malware apps have been discovered.[5] To our knowledge, research conducted to combat this type of abuse is still in its infancy [221].

Aiming for credible sources, a recommender usually chooses to crawl only from repositories with a considerably large number of stars and/or forks, or whatever criterion a recommender uses for selecting its training set (✔) [182]. In this case, an attacker may need to disguise a planted project by falsifying this information, e.g., by starring and forking with forged accounts (✗). Thus, there is a need for mechanisms to detect this type of fabrication (✔).

To penetrate a system, the introduction of a library is just the first step. The second step is to trick developers into invoking the library by calling its functions. Thus, an adversary needs also to steer API or code recommenders (✗), e.g., the ones recently published [83, 181, 233]. We conjecture that a technique similar to what described in Section 9.1.2 can be used for injecting toxic APIs.

9.2 Adversarial attacks to API RSSEs

9.2.1 Push attacks to API and code snippet RSSE

During the development process, programmers may look for and embed relevant APIs or code snippets useful for their tasks [181]. While this is a common practice as it helps increase productivity [160], it also poses security concerns.

Let us consider a developer who is using a tailored tool to search for snippets relevant to her context. Such a type of system has been chosen as it represents a typical scenario in software development where the recommendation needs to be learned from public reposito-

[4] We mined library usage directly from *pom.xml* files.
[5] https://zd.net/3bg3CK9

ries, and the adoption of a malicious API or code snippet may put the software system under development at risk [22]. In this respect, we anticipate two scenarios in which the developer is trapped by hostile attempts as follows: She is provided with either *(i)* APIs coming from legitimate libraries, which may trigger disruptions/fatal errors if being executed in certain contexts/under some usage scenarios; or *(ii)* intentionally harmful APIs embedded in a fake library.

To illustrate the first scenario, i.e., normal APIs causing fatal errors, we refer to the example[6] in Listing 9.1.

Listing 9.1 A snippet seen as harmless, but actually harmful.

```
1  import java.io.OutputStream;
2  import java.net.Socket;
3
4  public class Test {
5    public static void main(String[] args) throws Exception {
6          Test test = new Test();
7          test.debug("hello");
8    }
9    public void debug(String msg) throws Exception {
10         String s = "/usr/bin/logger ";
11         Runtime r = Runtime.getRuntime();
12         if (System.getProperty("os.name").equals("linux")) {
13               r.exec(s + msg);
14         } else {
15               Socket socket = new Socket("loghost", 514);
16               OutputStream out = socket.getOutputStream();
17               out.write(new byte[] {0x2A, 0x2F, 0x72, 0x2E, 0x65, 0x78, 0x65, 0x22, 0x72, 0x6D, 0x22, 0x3B, 0x2F,
                      0x2A });
18               out.write(msg.getBytes());
19         }
20   }
21 }
```

The Java-written snippet looks harmless as first sight, however, once being executed it has a severe consequence on the hosting client. To be concrete, the bytecode `0x2A, 0x2F,..., 0x2F, 0x2A` (Line 17) corresponds to the following string: `*/r.exe"rm";/*`. The implication of the `out.write()` method with the string as parameter in the Windows operating system is the deletion of random files.[7] In this case, while `out.write()` is a *native method* which comes from `java.io.OutputStream` – a mainstream library – it still induces a detrimental effect in the hosting platform.

The second scenario is when the developer is provided with intentionally harmful APIs embedded in a fake third-party library. As an example, Listing 9.2 depicts a third-party

[6]This code originates from the following blog post: https://bit.ly/31R760l

[7]The snippet is dangerous, and thus we advise against running it. Detailed explanations can be found in the original blog post.

library with the `FileManager` class, which wraps the malicious code in Listing 9.1 using the `writeLog()` declaration. Though the name has nothing to do with the code, it makes the declaration appear more legitimate, helping disguise the intent better [187]. Eventually, the library is compiled and published as a JAR file.

Listing 9.2 A third-party library to wrap malicious code.

```
1  package tools;
2  import java.io.IOException;
3  import java.io.OutputStream;
4  import java.net.Socket;
5  public class FileManager {
6    public void writeLog(String msg) throws Exception {
7      String s = "/usr/bin/logger ";
8      Runtime r = Runtime.getRuntime();
9      if (System.getProperty("os.name").equals("linux")) {
10       r.exec(s + msg);
11     } else {
12       Socket socket = new Socket("loghost", 514);
13       OutputStream out = socket.getOutputStream();
14       out.write(new byte[] { 0x2A, 0x2F, 0x72, 0x2E, 0x65, 0x78, 0x65,
15         0x22, 0x72, 0x6D, 0x22, 0x3B, 0x2F, 0x2A });
16       out.write(msg.getBytes());
17     }
18   }
19 }
```

Listing 9.3 is the new version of the project in Listing 9.1, however it is rewritten by means of the library, which is embedded through the `import tools.FileManager` directive (Line 3). The resulting snippet in Listing 9.3 is more compact, and it offers the same functionality as the project in Listing 9.1; however, all the intent is hidden in the library. The final usage pattern consists of only two API calls, i.e., `FileManager fm = new FileManager()` and `fm.writeLog()`. In this way, attackers render their attempt more practical by exposing the code in Listing 9.3 to the public, waiting for the developer to take the bait.

Listing 9.3 The new snippet using the third-party library.

```
1  import java.io.OutputStream;
2  import java.net.Socket;
3  import tools.FileManager;
4
5  public class Test {
6    public static void main(String[] args) throws Exception{
7        FileManager fm = new FileManager();
8        fm.writeLog("Kernel - Starting service");
9    }
10 }
```

Such a type of attack is effective, as it can be tailored for any specific purpose, e.g., creating a backdoor to render unauthorized access [288] once being successfully invoked. However, it also requires additional effort to plant the malicious library in OSS platforms and to trick the developer into calling it.

At the same time, both scenarios may appear to be unrealistic, as the possibility of encountering such snippets/APIs under normal circumstances is low, i.e., the developer would never come across the code when using recommender systems. However, by manipulating the training data in OSS platforms, e.g., GitHub, adversaries can boost up the snippets' visibility/popularity so that API recommenders will adopt and provide it to the developer. As such, the suggested snippet poses a potential danger to the software systems embedding it.

To reveal potential risks that maliciously handled training data might cause, we start from the assumption that some users have already adversarially-manipulated training sets. It is out of our scope to develop mechanisms to inject malicious snippets on crowdsourced repositories (e.g., on Stack Overflow) or make fake APIs become popular, e.g., by boosting their stars/forks and adding pages on Q&A forums. For the sake of presentation, in the rest of this chapter, we call an API causing negative effects or errors as *a malicious API*, regardless of its origin, i.e., whether it comes from a legitimate or from a fake library.
As mentioned before, although the code in Listing 9.1 is dangerous, it is unlikely that developers encounter something similar under normal circumstances. To expose the code to recommenders, attackers need to manipulate the training data by performing a *push attack*. In such a misdeed, they forcefully favor the targeted items by boosting their popularity. This increases the possibility of being discovered and thus, recommended by search engines.

We encounter the following challenge: "*How should a fake API be planted, so that it will be incorporated by recommendation engines?*" In fact, recommender systems rely heavily on similarity measures, i.e., they employ algorithms to search for similar artifacts, which are used to deduce recommendations [187]. This is the case not only for general-purpose recommender systems [213], but also for several RSSE [112, 167, 181, 223, 276, 298]. For instance, library recommenders search for libraries from the most similar projects in the training data [182, 183, 259]. Similarly, various systems for providing APIs and code also exploit a similarity measure [83, 177], or a kernel [103] to retrieve similar projects and snippets. More importantly, to produce recommendations, RSSE need to rely on OSS repositories, such as GitHub, or Maven, which are subject to changes from the public.

Let us imagine a scenario in which one increases the popularity of malicious APIs[8] by planting them to OSS projects, as many as possible. Fig. 9.2 illustrates the process in

[8] As stated in Section 9.2.1, we consider an API malicious if it causes fatal errors, no matter where it comes from, i.e., either a legitimate or a fake library.

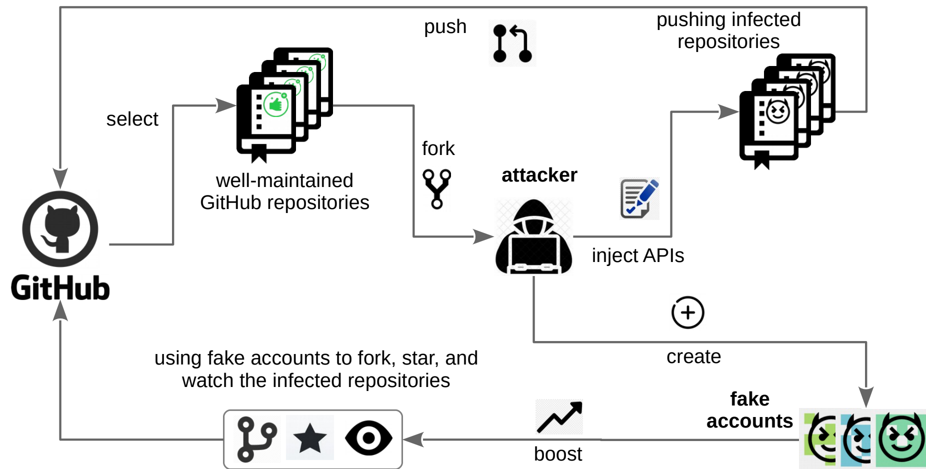


Fig. 9.2 The process of manipulating GitHub to promote malicious repositories.

which attackers may exploit to plant malicious data. First, well-maintained repositories are forked from GitHub, e.g., those that have good indicators in terms of stars, forks, or watchers. Afterward, the projects are injected with fake APIs, and then uploaded back to GitHub. Malicious repositories in GitHub are not scarce, but *dime a dozen* [221]. To boost up the popularity of an API pattern, the APIs can be seeded into a significant number of declarations, for each training project. Attackers may create fake accounts to star, fork, and watch malicious repositories to increase their credibility/visibility, thus exposing them better to search engines.[9]

9.2.2 Study design and planning

The *goal* of this study is to investigate the relevance of AML attacks for RSSE, and in particular of API and code snippet recommenders. The *quality focus* is the resilience of RSSE to AML attacks. The *perspective* is of researchers interested to improve the RSSE they develop. The *context* consists of state-of-the-art API and code snippet recommenders.

We aim to address the following research questions:

- **RQ**$_1$: *How well has the issue of AML in RSSE been addressed by the existing literature?* We perform a literature analysis to investigate whether there is already any effort devoted to study and deal with threats to RSSE originating from malicious data. Although our purpose is not to perform a complete, detailed systematic literature review, we followed consolidated guidelines for this kind of study in software engineering [129, 153].

[9] Such manipulation has been recently revealed https://zd.net/3bg3CK9.

- **RQ$_2$**: *To what extent are state-of-the-art API and code snippet recommender systems susceptible to malicious data?* First, we perform a qualitative analysis of the likely attack threats that could affect state-of-the-art API and code snippet recommenders. Then, based on their availability, we select three systems for our empirical evaluation, i.e., UP-Miner [276], PAM [83], and FOCUS [177, 181]. These are among the state-of-the-art approaches for API recommendations as they emerge from premier software engineering venues. We conjecture that an evaluation on these systems will help to shed light on the resilience of the majority of existing API recommenders.

9.2.2.1 Addressing RQ$_1$: Literature analysis

To achieve a good trade-off between the coverage of existing studies on AML in RSSE and efficiency, we defined the search strategy by answering the following four W-questions [292] ("*W*" stands for Which?, Where?, What?, and When?).

- *Which?* Both automatic and manual searches were performed to look for relevant papers from conferences and journals.

- *Where?* We conducted a literature analysis on premier venues in software engineering. In particular, there are nine conferences as follows: ICSE, ESEC/FSE, ASE, ICSME, ICST, ISSTA, ESEM, MSR, and SANER. Meanwhile, the following five journals were considered: TSE, TOSEM, EMSE, JSS, and IST.[10] The selection of conferences and journals was performed so to include mainstream venues, as well as specialized ones for which RSSE are relevant. The automatic search was done on the *SCOPUS* database.[11] We fetched all the papers published by a given edition (year) of a given venue (journal/conference) using the advanced search and export features.

- *What?* For each paper collected, its title and abstract were extracted using a set of predefined keywords. To cover more possible results, we used regular expressions for searching, e.g., depending on the terms we may use case sensitive queries.

- *When?* Since Adversarial Machine Learning is a recent research topic, we limit the search to the most five recent years, i.e., from 2016 to 2020.

The extraction process produced a corpus of 7,076 articles. Then, we performed various filtering steps to narrow down the search and look for those that meet our requirements. In

[10] We report the full name of all the venues as well as their corresponding acronyms in an online appendix https://bit.ly/3jUey4K.
[11] https://www.scopus.com/

particular, we are interested in studies dealing with recommender systems together with the relevant topics, i.e., Adversarial Machine Learning, API mining, and malicious attempts. Intermediate results, e.g., number of downloaded papers per venue, number of candidate papers per venue, are available in our online appendix [189].

9.2.2.2 Addressing RQ_2: Qualitative analysis and experiment on three RSSE

To address RQ_2, we looked at the same venues considered in RQ_1 to identify, over the period 2010–2020, API recommender systems as well as RSSE suggesting API code example snippets. We then qualitatively discuss, for each recommender, the working mechanism, and its potential risks.

Then, based on the tool availability as well as the characteristics of the tools, we select three of them, i.e., UP-Miner [276], PAM [83], and FOCUS [181]. To evaluate the resilience of UP-Miner, PAM, and FOCUS, we use a dataset containing Android apps' source code. We focused on Android apps because they entail a typical scenario in which an infection can cause unwanted consequences such as data leaks.

We made use of a dataset which was curated through our recent work [177], and the collection process is summarized as follows. First, we searched for open source projects using the *AndroidTimeMachine* platform [91], which retrieves apps and their source code from Google Play[12] and GitHub. Second, APK files are fetched from the Apkpure platform,[13] using a Python script. Third, the *dex2jar* tool [1] is used to convert the APK files into the JAR format. The JAR files were then fed as input for Rascal to convert them into the M^3 format [25]. We obtained a set of 2,600 apps with full source code. To simplify the evaluation, we inject APIs at the metadata level, i.e., after the data that has been parsed to a processable format. This is for experimental purposes only since, in reality, attackers need to seed data directly to projects and upload them to GitHub as shown in Fig. 9.2. However, in our evaluation we refrained from doing this to carefully follow ethical boundaries, as well as to avoid adversely impacting real-world systems.

We then inserted fake APIs into random projects and declarations, attempting to simulate real-world scenarios where APIs are dispersed across several declarations. Finally, the resulting data is parsed in two formats, i.e., ARFF files to be fed to UP-Miner and PAM, and a special file format for providing input to FOCUS. By an empirical evaluation, we realized that UP-Miner and PAM suffer from scalability issues, i.e., they cannot work on large datasets. Thus for evaluating them, we could only consider a subset consisting of 500

[12] https://play.google.com/
[13] https://apkpure.com/

apps. For FOCUS, the whole 2,600 apps are used since the system is capable of handling well a large amount of data.

There are the following parameters to consider when it comes to populating artificial projects.

- α is the ratio of projects injected with fake APIs (in percent, %).
- β is the ratio of methods in a project getting fake APIs (in percent, %).
- Ω is the number of fake APIs injected to each declaration.
- N is cut-off value for the ranked list of recommended items returned by a recommender system.

We experiment with the following sets of parameters: α={5%, 10%, 15%, 20%}, β={40%,-50%, 60%}, Ω={1,2}. The rationale behind the selection of these values is as follows. Concerning α, though having popular APIs is commonplace, it is difficult to rack up projects with malicious APIs, thus α is set a small percentage, i.e., α={5%, 10%, 15%, 20%}. In contrast, within a project, attackers have more freedom to embed APIs to declarations, therefore β is varied from 40%, 50%, to 60%. Finally, as explained in Section 9.2.1, the number of fake APIs should be kept low to make attacks more feasible, i.e., Ω={1,2}. We study how the calibration of the parameters affects the final efficiency, aiming to anticipate the extent to which the attacks are successful in the field.

We inspected the dataset produced as explained above, and counted 26,852 unique APIs in all the apps. Fig. 9.3 depicts the distribution of APIs in projects and declarations. The x-Axis and the y-Axis specify the number of projects and the number of declarations in which an invocation is seen, respectively. The dense cluster of points on the lower-left corner suggests that most of the APIs appear in less than 250 projects and 200 declarations. Meanwhile, a small fraction of them are invoked by a large number of projects and declarations, i.e., more than 2,000 and 10,000, respectively. Such a distribution has an impact on the α and β parameters.

The figure indicates that some APIs are extremely frequent. By looking inside the dataset, we see that `java/lang/StringBuilder/append(java.lang.String)` is the most popular API as it appears in 96.61% of the projects. Other invocations specific to Android apps, such as `android/view/View/getRight()`, are also very common in the dataset. The presence of very popular APIs gives us some hints on how to inject malicious APIs having the same names. We conjecture that, even if we embed an artificial API on a large number of projects, this can be seen as normal and thus, does not arouse developers' suspicion.

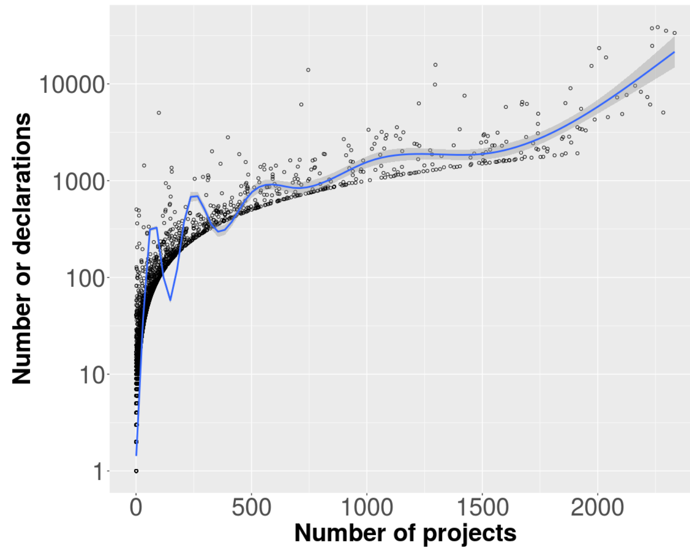


Fig. 9.3 Frequency of APIs in projects and declarations.

We conducted experiments using the ten-fold cross-validation methodology, i.e., the dataset with $|P|$ projects is divided into ten equal parts, and each experiment composed of ten train/test tasks. For each round of execution, one part is used for testing, while the other nine remaining parts are fed as training.

To measure the effectiveness of push attacks, we employ *Hit ratio HR@N* [18, 164], which is defined as the ratio of projects being provided with a fake API $|T|$ to the total number of testing projects $|P|$, i.e., $HR@N = |T| / |P|$, with N being the cut-off value for the ranked list. The metric is computed over the results of all the ten folds.

9.2.3 Results

This section reports the study results, addressing the research questions formulated in Section 9.2.2.

RQ$_1$: *How well has the issue of AML in RSSE been addressed by the existing literature?* Following the process in Section 9.2.2, we collected a corpus consisting of 7,076 documents coming from the considered conferences and journals in the five most recent years, i.e., from 2016 to 2020. By inspecting the corpus, we see that the number of papers varies among different venues, ranging from less than 20 to around 250 papers.

We are interested in studies about recommender systems and the related topics, i.e., Adversarial Machine Learning, API mining, and malicious attempts. From the collected corpus we narrowed down the scope by using five sets of keywords as shown in Table 9.5. For instance, for the REC set, there are the following keywords: "*recommendation*," "*recom-*

Table 9.4 Notable RSSE for mining APIs and/or code snippets (Listed in chronological order).

System	Venue	Year	Data source	Working mechanism	Potential risks	Avail.
UP-Miner [276]	MSR	2013	Microsoft Codebase	UP-Miner works on the basis of clustering, computing similarity at the sequence level, i.e., APIs that are usually found together using BIDE [275]. Finally, it clusters to group frequent sequences into patterns. Ⓢ Ⓒ	Similar to MAPO, as UP-Miner depends on BIDE, an attacker can inject malicious code in the training in projects disguised as similar to trick UP-Miner. In this way, it may recommend to developers harmful snippets.	✔
MUSE [167]	ICSE	2015	Java projects	MUSE automatically retrieves relevant API usages using static analysis techniques. It then ranks the resulting snippets employing code cloning detection as the similarity measure. Ⓢ	As it works by means of similarity among snippets, MUSE can be affected by malicious code embedded in similar projects planted in public platforms, e.g., GitHub.	✗
SALAD [191]	ICSE	2016	Google Play	SALAD learns API usages directly from bytecode extracted from Android apps. It relies on a hidden Markov model that predicts relevant API patterns according to their probabilities. Ⓢ	Since the most probable usages are retrieved as recommendations, a hostile user may plant malicious bytecode in Google Play to trick SALAD into recommending to developers.	✗
DeepAPI [101]	ESEC/ FSE	2016	GitHub	DeepAPI generates relevant API sequences starting from a natural language query. It employs an Encoder-Decoder RNN to encode words in context vectors. DeepAPI trains a model that encodes natural language annotations and API sequences. Afterwards, it uses the model to compute a list of API sequences. Ⓢ	An RNN bases also upon the notation of similarity, thus a malicious user can forge API sequence and textual annotation pairs to spoil the recommendations. First, she can remove or change part of the textual annotation or even worst, mix annotations and API sequences. Second, she may inject malicious APIs into sequences.	✗
PAM [83]	ESEC/ FSE	2016	GitHub	PAM defines a distribution over all possible API patterns in client code, based on a set of patterns. It uses a generative model to infer the most probable patterns. The system generates candidates by relying on the highest support first rule. Ⓢ	PAM recommends API calls that commonly appear in different code snippets. Thus, push and nuke attacks could modify the final ranking obtained by the tool, i.e., operating on terms' occurrences to favor or defame a certain API pattern.	✔
FINE-GRAPE [233]	EMSE	2017	GitHub	Relying on the history of the related files, FINE-GRAPE parses GitHub projects, discovers, and ranks the relevant API calls according to their history from every API version. Ⓢ	Manipulations that forge history of API calls in projects can pose a threat to the system. Another pitfall can be represented by the Java code that the tool parses to get relevant API patterns.	✗
FOCUS [177, 181]	ICSE	2019	GitHub, Maven, Google Play	FOCUS suggests APIs by encoding projects in a tensor and using a collaborative-filtering technique to deliver the list of APIs. Eventually, it mines APIs and snippets from similar projects with a graph representation. Ⓢ	The system is susceptible to poisoning attacks, i.e., an adversary can create fake similar projects containing toxic APIs and pose them as legitimate to deceiving FOCUS into recommending these calls/snippets.	✔
CodeKernel [103]	ASE	2019	Java projects	A graph-based representation is used to cluster similar API calls. Then, the system computes graph similarity by means of kernel functions. Code is selected according to designed ranking metrics, i.e., specificity and centrality. Ⓢ	Since its kernel functions work on graph structure, CodeKernel can be fooled by copycat Java APIs that closely resemble normal ones. Attackers can insert fake code in the graph embedding process to disguise them as similar.	✗
AuSearch [21]	SANER	2020	GitHub, Maven	AuSearch discovers API usages from GitHub by converting the input query into a GitHub query and searches for types of parameters that match with the ones contained in the query. It also relies on the Maven Package Search API algorithm to look for similar packages. Ⓢ	The tool is prone to poisoning attack with project containing malicious packages. Furthermore, an attacker can use JAR files obtained from fake projects to spoil the package analyzer module which works on top of the Maven Package Search API algorithm.	✗

Table 9.5 Keywords.

Acronym	Set of terms	Case-sensitive
REC	recommendation, recommender, recommendation systems	✗
API	API	✔
ADV	adversarial	✗
	AML	✔
ML	machine learning, machine-learning	✗
	ML	✔
MAL	malicious	✗

mender," or "*recommender systems*," which are popular terms used to refer to a system for providing recommendations.

	REC	ADV	ML	API	MAL
REC	506				
ADV	0	49			
ML	33	6	385		
API	51	3	29	370	
MAL	0	2	8	9	82

Fig. 9.4 Number of papers for the related topics.

Fig. 9.4 depicts a matrix whose each cell reports the number of papers that contain both the keywords in the corresponding column and row. Since it is a symmetric matrix, we list the numbers on the lower left part and leave the upper right part blank, for the sake of clarity.

Our search targets papers containing one of the following five combinations: either *(i)* REC and ADV; or *(ii)* ADV and ML; or *(iii)* ADV and API; or *(iv)* REC and MAL; or *(v)* API and MAL. We mark the associated cells using the green color, and carefully examine these papers. None of them matches with both sets of keywords REC and ADV, or REC and MAL. For the combinations REC and ML, ADV and API, API and MAL, we found six, three, and nine articles, respectively. By reading the abstract, we filtered out the ones being completely out of our interest. We ended up with only seven papers [23, 86, 131, 139, 172, 243, 289], discussed as follows.

Most of the resulting studies investigate API-based malware detection in the context of Android applications. For instance, Wu *et al.* [289] proposed an approach that relies on dataflow-related API-level features to detect malicious samples. Similarly, REVEALDROID [86] exploits a scalable, light-weight classification and regression tree classifiers (CART) to discover Android malware. API features are extracted directly from source code, i.e.,

by mining security-sensitive API calls. MKLDROID [172] is a framework for detecting malware and malicious code localization, and it integrates different semantic perspectives of apps, e.g., security-sensitive APIs, system calls, control-flow structures, information flows, in conjunction with ML classifiers. A recent work [139] analyzes vulnerabilities caused by the usage of advertising platform SDKs. Moreover, a static analysis tool named ADLIB has been developed to analyze advertising platform SDKs and detect vulnerable patterns that can be abused by advertisements. Bao *et al.* [23] proposed a study for detecting malicious behavior of malware that infects benign apps by mining sandboxes.

Singh *et al.* [243] proposed an approach to detection of behavior-based malware. Cuckoo sandboxes are inspected to extract three different primary features. After dedicated preprocessing steps, e.g., NLP, or decomposition, all the resulting features are integrated in the training set to develop malware classifiers using different machine learning algorithms, e.g., Random Forest, Decision Tree, or Support Vector Machines. Being conceived to automatically identify Server-based InFormation OvershariNg (SIFON) vulnerabilities, the Hush tool [131] employs a heuristic to analyze sensitive information excerpted from server-side mobile APIs. As a preliminary study, the system makes use of static program analysis to discover potential software vulnerabilities in the analyzed applications.

In summary, by thoroughly investigating the papers that match the keywords used to filter out irrelevant studies, we realize that all of them deal with malware detection in Android apps. Instead, we did not find any work related to potential threats and implications of adversarial attacks to API RSSE.

> **Answer to RQ$_1$.** So far, the issue of adversarial attacks to APIs and code snippet recommenders has not been adequately studied in major software engineering venues.

RQ$_2$: *To what extent are state-of-the-art API and code snippet recommender systems susceptible to malicious data?* We answer **RQ$_2$** by discussing a qualitative analysis of existing API recommender systems, and by presenting the results of the empirical evaluation, which has been performed by analyzing notable RSSE for mining APIs and code snippets.

9.2.3.1 Qualitative analysis

From the premier software engineering venues considered in **RQ$_1$** we selected work presenting techniques and tools recommending APIs and code snippets. Table 9.4 lists in chronological order the set of analyzed approaches.

For each system, by studying the used approach (**Working mechanism** column), we discuss its possible vulnerabilities (**Potential risks** column). The last column, namely **Avail.**,

specifies the availability of the replication package by each tool, i.e., either available (✔) or not available (✗).

Besides what is reported in the **Potential risks** column of Table 9.4, this section summarizes their main characteristics to highlight the risk of being exploited. Two features that make a system vulnerable to malicious attempts are namely *(i)* relying on data from the open-source for training, e.g., GitHub or Android markets; and *(ii)* the application of a similarity measure (marked with Ⓢ) or a clustering technique (marked with Ⓒ) to recommend APIs or code, as we detail below.

- All the systems leverage public data sources to function. While in principle RSSE can also be trained on closed-source, trusted repositories, in all those cases a realistic, broad learning makes it unavoidable to leverage large, open-source forges. Most of the considered RSSE are trained with repositories from GitHub, Maven, or the Microsoft Code Base, including UP-Miner [276], DeepAPI [101], PAM [83], FINE-GRAPE [233], AuSearch [21]. Since these sources are freely open for changes by the crowd, they are also exposed to malicious purposes. Other systems supporting Android apps, such as SALAD [191] or FOCUS [177] normally obtain data from Google Play, which enforces mechanisms to control submissions. Nevertheless, such a platform is not immune from manipulation, as this has been previously reported [46].

- Most of the approaches are based on a similarity measure/kernel to mine APIs and/or code snippets. In this way, an adversary can insert malicious APIs into similar projects, and pose them as legitimate to trick the systems into using the disguised project and eventually recommending API calls. In particular, the following systems work on the basis of a similarity algorithm: MUSE [167], FINE-GRAPE [233], FOCUS [177, 181], and CodeKernel [103]. UP-Miner [276] is not directly based on similarity, however, it relies on the BIDE algorithm [275], which works by mining from common patterns. Thus, it is prone to malicious code concealed in popular sequences.

- Similarly, the other systems that work on clustering techniques are also susceptible to adversarial attacks. In fact, besides the BIDE algorithm [275], MAPO and UP-Miner additionally employ clustering to group similar code sequences. In this way, attackers may populate fake projects to favor a particular code pattern containing malicious APIs. Lastly, approaches like FINE-GRAPE are prone to manipulations forging an artificial history of API calls in GitHub projects.

Altogether, it is evident that all the systems in Table 9.4 are potentially exposed to push attacks. They can be manipulated to favor a malicious API or snippet, so that it will be

suggested by the recommendation engine. As such, the systems in which the recommended code is embedded will suffer.

9.2.3.2 Empirical evaluation

To quantitatively investigate the extent to which the threats outlined in the previous qualitative analysis can actually occur, we perform an empirical evaluation on three among the systems in Table 9.4, i.e., UP-Miner [276], PAM [83], and FOCUS [181] using the collected dataset (see Section 9.2.2).

We selected these tools due to the following reasons. *First*, UP-Miner is a well-established recommender system, which has shown to outperform MAPO [298], one of the first systems for suggesting APIs. PAM has been proven to be more effective compared to MAPO and UP-Miner [83]. Meanwhile, FOCUS is the most recent approach and it obtains the best prediction performance if compared to both UP-Miner and PAM [177]. *Second*, the considered systems are also representative in terms of working mechanism (see Table 9.4). UP-Miner works based on clustering, while PAM mines API patterns that commonly appear in different snippets, and finally FOCUS exploits a collaborative-filtering technique, i.e., also based on a similarity algorithm, to retrieve APIs from similar projects. In this respect, we conjecture that an evaluation on the three systems could be generalizable to the remaining ones in Table 9.4. *Third*, the three tools have evaluation replication package available, i.e., being specified with (✔) in the **Avail.** column of Table 9.4. Such implementations enable us to run the experiments according to our needs.

The number of ranked items N is internal, i.e., it can be customized by developers. In contrast, α, β, and Ω are external since they can be tuned by adversaries to make their attacks more effective. The increase of α, β, and Ω is related to the extent to which attackers add fake APIs to more projects. In the evaluation, we experiment with different configurations by varying these parameters to analyze which settings bring more perturbed outcomes.

Table 9.6 shows the hit ratio HR@N obtained by the recommendation results of UP-Miner. It is evident that the system is considerably affected by the crafted training data, i.e., it recommends the fake APIs to several projects, depending on the input parameters α, β, and Ω. Even with a small ratio of infected projects e.g., α=5%, UP-Miner recommends the artificial APIs to hundreds of projects. For instance, considering Ω=1, the hit ratio HR@5 is 0.078 and it increases to 0.119 when N=20. When the fake API is injected to more declarations (increasing β), the hit ratio gradually improves: given that α=20%, β=40%, we get HR@5=0.262, while with β=60% HR@5 is 0.295. The same trend can be seen with other values of α and β. The hit ratio is proportional to α – the ratio of infected projects. In particular, the attacks become most successful when α=20% and β=60%, i.e.,

Table 9.6 Hit ratio for the recommendations by UP-Miner.

		Ω=1				Ω=2			
	α	**5%**	**10%**	**15%**	**20%**	**5%**	**10%**	**15%**	**20%**
β = 40%	**HR@5**	0.078	0.141	0.200	0.262	0.005	0.094	0.021	0.032
	HR@10	0.088	0.179	0.252	0.336	0.031	0.518	0.073	0.110
	HR@15	0.119	0.221	0.313	0.397	0.072	0.990	0.139	0.192
	HR@20	0.119	0.226	0.317	0.401	0.104	0.169	0.247	0.327
β = 50%	**HR@5**	0.098	0.169	0.213	0.266	0.000	0.047	0.017	0.032
	HR@10	0.114	0.188	0.256	0.331	0.031	0.424	0.065	0.106
	HR@15	0.130	0.235	0.326	0.409	0.083	0.115	0.156	0.209
	HR@20	0.130	0.235	0.326	0.409	0.109	0.193	0.273	0.336
β = 60%	**HR@5**	0.093	0.174	0.239	0.295	0.015	0.014	0.021	0.028
	HR@10	0.193	0.356	0.282	0.356	0.041	0.056	0.065	0.102
	HR@15	0.231	0.401	0.321	0.401	0.078	0.127	0.147	0.196
	HR@20	0.235	0.409	0.326	**0.409**	0.098	0.193	0.265	0.331

Table 9.7 Hit ratio for the recommendations by PAM.

		Ω=1				Ω=2			
	α	**5%**	**10%**	**15%**	**20%**	**5%**	**10%**	**15%**	**20%**
β = 40%	**HR@5**	0.048	0.098	0.148	0.198	0.044	0.090	0.140	0.198
	HR@10	0.050	0.100	0.150	0.200	0.048	0.098	0.148	0.198
	HR@15	0.050	0.100	0.150	0.200	0.050	0.100	0.150	0.200
	HR@20	0.050	0.100	0.150	0.200	0.050	0.100	0.150	0.200
β = 50%	**HR@5**	0.048	0.098	0.148	0.198	0.048	0.098	0.148	0.198
	HR@10	0.500	0.100	0.150	0.200	0.048	0.098	0.148	0.198
	HR@15	0.500	0.100	0.150	0.200	0.050	0.100	0.150	0.200
	HR@20	0.050	0.100	0.150	0.200	0.050	0.100	0.150	0.200
β = 60%	**HR@5**	0.048	0.098	0.148	0.198	0.048	0.096	0.146	0.196
	HR@10	0.050	0.100	0.150	0.200	0.048	0.098	0.148	0.198
	HR@15	0.050	0.100	0.150	0.200	0.050	0.100	0.150	0.200
	HR@20	0.050	0.100	0.150	0.200	0.050	0.100	0.150	**0.200**

Table 9.8 Hit ratio for the recommendations by FOCUS.

		Ω=1				Ω=2			
	α	**5%**	**10%**	**15%**	**20%**	**5%**	**10%**	**15%**	**20%**
β = 40%	**HR@5**	0.012	0.021	0.028	0.039	0.009	0.025	0.034	0.034
	HR@10	0.029	0.053	0.078	0.115	0.026	0.055	0.087	0.106
	HR@15	0.032	0.068	0.101	0.145	0.031	0.070	0.106	0.141
	HR@20	0.038	0.081	0.119	0.173	0.037	0.083	0.119	0.168
β = 50%	**HR@5**	0.014	0.036	0.050	0.067	0.017	0.036	0.048	0.063
	HR@10	0.033	0.073	0.105	0.140	0.033	0.073	0.105	0.139
	HR@15	0.040	0.081	0.126	0.164	0.038	0.083	0.123	0.164
	HR@20	0.046	0.089	0.138	0.192	0.044	0.090	0.136	0.181
β = 60%	**HR@5**	0.023	0.051	0.072	0.028	0.094	0.047	0.070	0.097
	HR@10	0.040	0.083	0.123	0.171	0.038	0.080	0.120	0.160
	HR@15	0.045	0.093	0.138	0.190	0.041	0.088	0.131	0.173
	HR@20	0.048	0.099	0.149	**0.203**	0.047	0.096	0.139	0.185

HR@20=0.409. Given that Ω=2, we obtain a comparable outcome to that with Ω=1. To summarize, we conclude that *UP-Miner is prone to adversarial attacks since it suggests malicious APIs to developers.*

Results for PAM are reported in Table 9.7. Similar to UP-Miner, PAM is also not immune from attacks as it recommends to developers the fake APIs. However, PAM is less susceptible to manipulations compared to UP-Miner since we get lower hit ratios. The maximum HR@N is 0.200, obtained when α=20% and β=60%. Varying N as well as Ω seems to have a

negligible impact on the final results. This can be explained by considering the underlying algorithm of PAM, which retrieves APIs that appear more frequently. As the two APIs are planted together, they will be recommended commonly as a pattern. Therefore, adding one API does not significantly affect the final hit ratio. Overall, the results in Table 9.7 suggest that *the use of PAM may be threatened by malicious attempts concealed in training data.*

Finally, we investigate how negative the effect might be for developers using FOCUS as their recommender, given that the training data has been manipulated. Note that for evaluating the system, we use the whole dataset with 2,600 Android apps. The maximum hit ratio is 0.203 and obtained when α=20%, β=60%. In other words, users of the system are likely to be recommended with the manipulated code. The hit ratios for FOCUS recommendations are comparable to PAM, although *(i)* in this case the training set is larger, and hence comparable values of α and β mean a larger effort by the attacker; *(ii)* since FOCUS provides to developers both APIs and snippets, the recommendation of a malicious API pattern may induce a serious consequence on the receiving client. In summary, also for FOCUS *the seeded data has an adverse effect, i.e., the tool is tricked into providing developers with the fake/toxic APIs, as well as code snippets.*

Overall, RSSE could recommend malicious APIs or code snippets if being trained with malicious data. All three API RSSE are affected by adversarial attacks. Understanding the technical motivations behind such results is not in the scope of this chapter. However, according to the performed analysis, UP-Miner and PAM provide fake APIs for a considerably large number of projects, even when only 10% of the training is manipulated. Though FOCUS is less prone than UP-Miner, the consequences caused by its recommendations can be devastating as the tool supplies also code snippets.

Answer to RQ$_2$. Toxic training data can pose a prominent threat to the resilience of state-of-the-art RSSE, including UP-Miner, PAM, and FOCUS.

9.2.4 Threats to validity

Threats to *construct validity* concern the relationship between theory and observation. In particular, they are related to the extent to which the simulated feeding of fake APIs or malicious snippets reflects a realistic AML attack scenario. We simply wanted to experiment with how a given percentage of projects with malicious snippets or APIs would affect the recommender's result. Evaluating the feasibility of a real attack is beyond the scope of the presented work.

Threats to *internal validity* are the confounding factors internal to our study that might have an impact on the results. One possible threat is the choice of venues, i.e., there may be

AML-related research, as well as relevant RSSE to study, published in venues that we did not consider, for instance, security conferences such as the USENIX Security Symposium.[14] Nevertheless, as shown in Section 9.2.2, we selected all major and topic-relevant software engineering journals and conferences, where RSSE are more likely to be found.

To evaluate the three API recommender systems, we used the original implementations of PAM[15] and FOCUS[16] made available by their authors. Since the original replication package of UP-Miner is no longer in use, we exploited the remake done by the authors of PAM. To minimize the threats that may affect the internal validity, we adopted the same experimental settings used in the original papers [83, 181, 276]. Also, we ran our experiments with different systems configurations to evaluate their impact on the effects of AML attacks.

Threats to *external validity* are related to the generalizability of our results. Such generalizability concerns *(i)* on the one hand the recommenders on which the experimentation has been carried out (conclusions may or may not apply to other recommenders); and *(ii)* on the other hand the considered dataset. For the former, in principle at least all the recommenders in Table 9.4 are likely to treat legitimate and malicious APIs and snippets similarly. For the latter, both the push attacks and our evaluation are generalizable also to other languages, such as Python.

In the following, we first discuss the feasibility of the RSSE attacks. Afterward, we overview possible defense mechanisms based on existing techniques, which are expected to be applicable for protecting RSSE against AML attacks.

9.2.5 Discussions

9.2.5.1 Feasibility of the attacks

RSSE rely on third-party data sources, i.e., they are usually fed with data from OSS repositories, which are open for changes including the phony ones. The probability that RSSE come across toxic sources cannot be ruled out. There are precedents where thousands of repositories with malware apps have been unearthed in GitHub [221], and they might be only *the tip of the iceberg*.

Though RSSE attempt to crawl training data from repositories considered to be *credible* [182], e.g., having a significant number of stars, forks, or watchers, unfortunately, this cannot help them completely evade toxic repositories. These metrics, however, can be falsified as attackers use fake accounts to star, fork, and watch the malicious repositories, making

[14] https://www.usenix.org/conference/usenixsecurity20
[15] https://github.com/mast-group/api-mining
[16] https://github.com/crossminer/FOCUS

them appear more legitimate. Such a trick has been recently revealed, where several fake accounts are used to reciprocally endow their malicious repositories.[17] To our knowledge, research conducted to fight this type of abuse is still in its initial phase [94]. Thus, techniques to conduct attacks are known, and there are at least examples of fake repositories, albeit being created for other purposes rather than for attacking RSSE.

Adversaries may also have different ways to camouflage their hostile intent. Apart from wrapping malicious code in a single API call (see Section 9.2.1), they might disguise it with a typosquatting name [187], i.e., one that closely resembles a popular API. In this case, developers would adopt the disguised API/snippet without the least delay, once it is provided by the recommendation engine.

Finally, the results obtained in RQ_2 suggest that even with a small amount of artificial training data, hit ratios are always larger than 0, implying that clients are being provided with the fake APIs. Altogether, we see that API recommender systems are likely to be exploited, and in this way they inadvertently become a *trojan horse*, causing havoc to software systems.

9.2.5.2 On the quest for potential countermeasures

While the topic of AML has been studied in different domains, there is no work dealing with adversarial attacks to RSSE (see Section 9.2.3). Also, there exist no concrete countermeasures that can be instantly applied to protect RSSE against attacks.

In the following, we discuss possible defense mechanisms from two perspectives, namely *(i)* internal view, i.e., design of recommender systems; and *(ii)* external view, i.e., techniques to detect and protect against hostile attempts. Concerning the former, we study counteractions proposed for generic recommender systems in the hope of customizing them for RSSE. Concerning the latter, we look for feasible ways to recognize and seize malicious APIs.

To **minimize the effect of manipulated profiles**, model-based algorithms are of great use as they can be applied to cluster similar profiles (OSS projects) into *aggregate segments* [164]. Though these techniques cannot entirely isolate malicious projects, this aims to lessen the prevalence of projects with abnormal behaviors, i.e., they will not be seen as similar to any active projects, i.e., the ones under development. In this way, such a method reduces the possibility that RSSE select and incorporate bogus data, thereby avoiding attacks. The method can be applied to defend RSSE that work based on a similarity or collaborative-filtering technique, such as UP-Miner [276], MUSE [167], FOCUS [177], or CodeKernel [103]. Nevertheless, it requires the redesign of the whole systems' underpinning building blocks, and thus, it is not easy to conduct.

[17] https://zd.net/3bg3CK9

Adversarial attacks can be counteracted using **anomaly detection techniques**, i.e., recognizing malicious API patterns before they are recommended to developers. For instance, a statistical process control strategy [164] has been used to identify suspicious items by examining two parameters, namely items' distribution density and average ratings. By referring to RSSE, we can think of detecting anomalies from the distributions of APIs in projects and declarations. This, however, necessitates careful analyses to avoid false positives and false negatives. As shown in Section 9.2.2, the distributions of APIs may span in a wide range, i.e., there are not only extremely popular APIs but also rare ones. Thus, being based on a radical pattern, e.g., a too popular or too rare one, the detection of malicious APIs may not succeed in every case. A more tailored approach is to tell malicious/benign API sequences apart by monitoring a certain set of third-party libraries using supervised classifiers [80]. Such an approach, however, has its limitation as follows. Though it can detect malicious sequences consisting of APIs from a specific set of libraries, it may not be applicable to patterns with a few APIs coming from a less popular library.

Profile classification can also help increase the resilience of RSSE against malicious attacks [50]. First, it is necessary to build a training set consisting of both authentic projects and fake projects that are generated following an attack model. Attribute-reduction techniques may be used to reduce the number of features needed to represent the dataset. Afterward, supervised classifiers are trained on the resulting dataset to classify real and fake projects, aiming to detect the injection of malicious data. This technique works more effectively when we have enough training data by taking into consideration different ways of populating fake projects.

A promising field is enhancing the robustness of ML-based systems. Unless profile classification strategy, those approaches aim at improving the underlying algorithm by exploiting feature-space attack models [291]. Even though several approaches propose robust models to support different tasks [260, 115], there is still room for improvements, e.g., handling realizable attacks or moving to generalized robustness.

In conclusion, we believe that a hybrid security model, consisting of countermeasures pertaining to both an internal (design) and external view, is likely to contribute to the robustness and resilience of RSSE towards adversarial attacks. While internal countermeasures allow RSSE to avoid falsified or suspicious data sources, defense mechanisms based on external view help RSSE detect malicious intent hidden in API patterns before recommending them to developers.

9.3 Conclusion

In recent years, we have witnessed a dramatic increase in the application of Machine Learning algorithms in several domains, including the development of recommender systems for software engineering (RSSE). While researchers focused on the underpinning ML techniques to improve recommendation accuracy, little attention has been paid to make such systems robust and resilient to malicious data. By manipulating the algorithms' training set, i.e., large open-source software (OSS) repositories, it would be possible to make recommender systems vulnerable to adversarial attacks. To rise the attention of the community on this topic, This chapter present two initial investigations that spot several vulnerabilities of two main RSSEs, i.e., TPLs and API recommenders.

First, we presented an initial investigation of the topic of AML in RSSE. While RSSE suggesting libraries and API calls have gained momentum, we showed how they can be susceptible to adversarial attacks with bogus input data. By spoiling the tools' recommendation list, the attack paves the way for unauthorized access to software clients.

In the second part of the chapter, we show that while API recommender systems become more effective at providing relevant recommendations, little attention has been paid to make them robust and resilient to adversarial attempts concealed in training data. First, through literature analysis, we realized that no studies have been conducted to investigate the abuse of deliberately forged data to spoil API recommenders' outcomes and conceive suitable counteractions.

An investigation into the working mechanism of existing API/code snippet recommender systems reveals their vulnerability to hostile attempts. Then, an empirical evaluation on three state-of-the-art API/code snippet recommenders further confirms our conjecture: all of them are exposed to malicious data, paving the way for unscrupulous exploitation.

Chapter 10

Conclusion

To face the abundance of miscellaneous sources of information, RSs are becoming widespread in different application domains to provide personalized items given the active context. In the software engineering domain, we witness the proliferation of automated approaches to support the development of complex software projects. Although RSs facilitate the developers' daily activities, there is an urgent need to simplify their development and customization by defining a precisely curated and organized core set of concepts and practices. In such a way, non-expert users can make use of complex models and algorithms that usually require a deep knowledge of the application domain. This dissertation presents a series of RSs specifically designed to recommend API function calls, categorize software repositories, and assist users during the specification of various modeling artifacts. As a further step, we elicit common components and processes to create a dedicated feature model and domain-syntax language, aiming at covering the specification of any class of RSs. Being built on top of such concepts, we propose an MDE-based tool to design and deploy custom RSs based on different strategies. We summarize the contributions in the RS domain in Section 10.1. All the publications are listed in Section 10.2, including the work published and under review not directly discussed in this dissertation. Eventually, Section 10.3 describes ongoing works and possible future work in the domain.

10.1 Summary of the contributions

This section summarizes the contributions of this dissertation in the RS domain by considering the challenges and the research objective presented in Section 1.1 and Section 1.2 respectively. The main contributions of this work are summarized as follows:

Recommending relevant API function calls. In Chapter 4, we presented two approaches to provide the developers with relevant API function calls given the active context, i.e., FOCUS and LUPE. FOCUS exploits a context-aware collaborative-filtering system to assist developers in selecting suitable API function calls and usage patterns. A thorough evaluation has been conducted *(i)* on an Android dataset to study the approach's performance, and *(ii)* in a user study with 16 participants to assess the perceived usefulness of FOCUS recommendations.

LUPE recommends *cohesive* APIs by relying on a sequence-to-sequence ML translation technique. Through an empirical evaluation conducted on data from the Android domain, we showed that the proposed tool obtains a satisfying prediction performance on real-world datasets, thereby outperforming two state-of-the-art baselines, GAPI [145] and FACER [9].

Recommending GitHub topics. Chapter 5 presented MNBN, an initial approach to recommend a set of featured topics given a software project endowed with a corresponding README file. The tool is based on a probabilistic machine learning network, the Naïve Bayesian classifier. We encode the relevant information about repositories using the TF-IDF weight scheme. After the training phase, the approach provides the user with a list of featured topics related to the project. Even though the conducted evaluation showed that MNBN is capable of providing decent topics, it suffers from some degradations of performances in terms of accuracy when an unbalanced dataset is considered. To overcome this limitation, we conceived HybridRec, a hybrid recommender system working on top of a stochastic network and a collaborative-filtering technique to recommend topics. We performed an empirical evaluation on real-world datasets to study HybridRec by comparing it with state-of-the-art toots. The results showed that the newly conceived approach improves our former recommender systems substantially. More importantly, we demonstrated that HybridRec can increase its prediction performance on well-curated data sources.

Assisting modelers during the specifications of modeling artifacts. In Chapter 6, we first introduced MemoRec, a novel approach that uses a context-aware collaborative filtering technique to support the modeler in completing the specification of a metamodel. By encoding metamodels and their contents in four different schemes, we built rating matrices and applied a syntactic-based similarity function to predict missing items, i.e., classes and structural features. An evaluation on two independent datasets, i.e., $\mathbf{D}_1$ and $\mathbf{D}_2$, and four encoding schemes, i.e., SE_s, IE_s, SE_c, and IE_c, exploiting ten-fold cross-validation demonstrates that the tool is able to provide decent recommendations. The second part of the chapter discussed MORGAN, a model recommender system that has been built on top of a graph

kernel similarity, with the ultimate aim of supporting modelers in specifying three different modeling artifacts, i.e., Ecore metamodels, XMI models, and JSON schema. An empirical evaluation of five real-world datasets demonstrated that MORGAN is applicable in different application domains, even though we experiment the scalability issue when a larger training set is considered.

Engineering the design, customization, and deployment of RSs. With the aim of simplifying the development of RSs from scratch, we first report the experience gained in the CROSSMINER EU project in Chapter 7, where a set of recommender systems has been conceived to assist software programmers in different phases of the development process. The systems provide developers with various artifacts, such as third-party libraries, documentation about how to use the APIs being adopted, or relevant API function calls. To develop such recommendations, various technical choices have been made to overcome issues related to several aspects including the lack of baselines, limited data availability, decisions about the performance measures, and evaluation approaches. Based on a set of lessons learned, we elicited recurrent patterns and concepts that occurs while specifying these systems. Afterward, Chapter 8 presented LEV4REC as a workable solution to assist developers that do not have strong experience in designing and programming recommender systems. Our approach is a MDE environment to foster an RS's design, configuration, and deployment from scratch using such a cutting-edge paradigm. LEV4REC is flexible and extensible as it relies on three core techniques, i.e., feature model, metamodel, and Acceleo templates. Starting from a feature model, RS designers can specify the system's features and then progressively enrich a configuration model automatically generated out of the selected features. Once the RS configuration has been refined, the system employs a model-driven code generator to produce the actual code of the specified RS. LEV4REC allows developers to refine the produced system by experimenting with different algorithms, experimental settings, and evaluation metrics. We evaluated the approach empirically by reimplementing two existing RSs that rely on different algorithms, i.e., collaborative filtering technique and feed-forward neural network. To discuss qualitative aspects of the proposed approach, we interviewed five domain experts by employing the focus group methodology, widely used in software engineering, to gather feedback on the benefits and limitations of the proposed approach.

Investigating adversarial attacks to RSSEs. In Chapter 9, we presented two empirical studies, aiming to investigate how existing RSSEs are prone to poisoning attacks. In the first study we showed how TPL RSSEs can be susceptible to adversarial attacks with bogus input

data. By spoiling the tools' recommendation list, the attack paves the way for unauthorized access to software clients. Our findings have been confirmed by analyzing another class of RSSEs, i.e., approaches that recommends API function calls. The conducted study on three different systems revealed that *(i)* the SE community underestimates the issues and *(ii)* notable state-of-the-art approaches can be fooled by injecting fake data.

10.2 Publications

This section lists all the papers that have been published during my three years as a Ph.D. student at the University of L'Aquila. The publications appear in workshop and conference proceedings, as well as journals. Besides these papers, I also present additional work that addresses various challenges using similar strategies but in different domains. For those that are not directly discussed in the dissertation, a disclaimer is added to highlight my contribution. The publications are listed in reverse chronological order.

Journals

J11 - Sas, C., Capiluppi, A., Di Sipio, C., Di Rocco, J., and Di Ruscio, D. (2023). GitRanking: A Ranking of GitHub Topics for Software Classification using Active Sampling, Software: Practice and Experience, 2023. ***In this work, we present a semi-automatic approach to produce a GitHub taxonomy. However, it is not introduced in any chapter of this dissertation.***

J10 - Di Sipio, C., Di Rocco, J., Di Ruscio, D., Nguyen, P. T., *MORGAN: An intelligent modeling assistant based on kernel similarity and graph neural networks*, Journal of Software and Systems Modeling, 2023, DOI: 10.1007/s10270-023-01102-8. ***This work was presented in Chapter 6.***

J9 - Nguyen, P. T., Di Sipio, C., Di Rocco, J., Di Penta, M., and Di Ruscio, D., *Fitting Missing API Puzzles with Machine Translation Techniques*, Journal of Expert Systems With Applications, 2022, DOI: 10.1016/j.eswa.2022.119477 ***This work was presented in Chapter 4.***

J8 - Nguyen, P.T., Di Rocco, J., Rubei, R., Di Sipio C., and Di Ruscio, D., *DeepLib: Machine translation techniques to recommend upgrades for third-party libraries*, Expert Systems with Applications, Volume 202, 2022, 117267, ISSN 0957-4174, DOI: 10.1016/j.eswa.2022.117267. ***This work employs a similar neural network proposed in Chapter 4 to recommend libraries migration. However, it is not presented in any chapter of this dissertation.***

J7 - Rubei, R., Di Ruscio, D., Di Sipio, C, Di Rocco J., and Nguyen, P.T., *Providing upgrade plans for third-party libraries: a recommender system using migration graphs.* Applied Intelligence, 12000–12015 (2022). DOI: 10.1007/s10489-021-02911-4. ***This work proposed a recommender systems for upgrading third-party libraries. However, it is not presented in any chapter of this dissertation.***

J6 - Di Rocco, J., Di Ruscio, D., Di Sipio, C. Nguyen, P.T., and Pierantonio, A., *MemoRec: a recommender system for assisting modelers in specifying metamodels.* Software and Systems Modeling (2022). DOI: 10.1007/s10270-022-00994-2. ***This work was presented in Chapter 6.***

J5 - Di Rocco, J., Di Ruscio, D., Di Sipio, C., Nguyen, P.T., and Rubei, R., *HybridRec: A recommender system for tagging GitHub repositories.* Applied Intelligence (2022). DOI: 10.1007/s10489-022-03864-y. ***This work was presented in Chapter 5.***

J4 - Di Rocco, J., Di Ruscio, D., Di Sipio, C., Nguyen, P.T., and Rubei, R., *Development of recommendation systems for software engineering: the CROSSMINER experience*. Empirical Software Engineering, 26(4):1–40, 2021. DOI: 10.1007/s10664-021-09963-7. ***This work was presented in Chapter 7.***

J3 - Nguyen, P.T., Di Rocco, J., Di Sipio, C., Di Ruscio, D., and Di Penta, M., *Recommending api function calls and code snippets to support software development.* IEEE Transactions on Software Engineering, pages 1–1, 2021, DOI: 10.1109/TSE.2021.3059907. ***This work was presented in Chapter 4.***

J2 - Duong, L. T., Nguyen, P. T., Di Sipio, C., and Di Ruscio, D., *Automated fruit recognition using EfficientNet and MixNet*, Computers and Electronics in Agriculture, Volume 171, 2020, 105326, ISSN 0168-1699, DOI: 10.1016/j.compag.2020.105326. ***This work exploits benchmarking models for the classification task.***

J1 - Rubei, R., Di Sipio, C., Nguyen, P.T., Di Rocco, J., and Di Ruscio, D., *PostFinder: Mining Stack Overflow posts to support software developers*, Information and Software Technology, Volume 127, 2020,106367, ISSN 0950-5849, DOI: 10.1016/j.infsof.2020.106367. ***This work is recommender systems for retrieving Stack Overflow posts. I contributed in this work by tuning the underpinning engine.***

Conferences

C10 - Nguyen P.T., Rubei R., Di Rocco J, Di Sipio C., Di Ruscio D., Di Penta M, *Dealing with Popularity Bias in Recommender Systems for Third-party Libraries: How far Are*

We? In proceedings of the 20th International Conference on Mining Software Repositories (MSR 2023), DOI: 10.1109/MSR59073.2023.00016. ***This work presented an initial investigation on popularity bias in RSSEs. However, it is not presented in any chapter of this dissertation.***

C9 - Di Rocco J., Di Sipio, C, Nguyen, P.T., Di Ruscio, D, and Pierantonio, A. 2022. *Finding with NEMO: a recommender system to forecast the next modeling operations*. In Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems (MODELS '22), DOI: 10.1145/3550355.3552459. ***This work exploits a similar neural network presented in Chapter 4 to forecast the next modeling operations. However, it is not presented in any chapter of this dissertation.***

C8 - Di Sipio C. 2022. *Automating the design of recommender systems: from foundational aspects to actual development*. In Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings (MODELS '22). DOI: 10.1145/3550356.3552376 ***This work has been published as a part of ACM student research competition at MODELS2022. Part of the content was presented in Chapter 8.***

C7 - Rubei, R., Di Sipio, C., Di Rocco, J., Di Ruscio, D., and Nguyen, P.T., *Endowing third-party libraries recommender systems with explicit user feedback mechanisms*, 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2022, pp. 817-821, DOI: 10.1109/SANER53432.2022.00099. ***This work presents an initial investgation on integrating user feedback in RSSE. However, it is not presented in any chapter of this dissertation.***

C6 - Nguyen, P.T., Di Sipio, C, Di Rocco, J., Di Penta, M., and Di Ruscio, D., *Adversarial Attacks to API Recommender Systems: Time to Wake Up and Smell the Coffee?*, 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2021, pp. 253-265, DOI: 10.1109/ASE51524.2021.9678946. ***This work was presented in Chapter 9.***

C5 - Di Sipio, C., Di Rocco, J., Di Ruscio, D. and, Nguyen, P.T., 2021. *A Low-Code Tool Supporting the Development of Recommender Systems*. In Fifteenth ACM Conference on Recommender Systems (RecSys '21). Association for Computing Machinery, New York, NY, USA, 741–744. DOI: 10.1145/3460231.3478885. ***In this work, we proposed the first version of LEV4REC. The extended version of the tool is presented in Chapter 8.***

C4 - Di Rocco, J., Di Sipio, C., Di Ruscio, D., and Nguyen, P.T., *A GNN-based Recommender System to Assist the Specification of Metamodels and Models*, 2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS), 2021, pp. 70-81, DOI: 10.1109/MODELS50736.2021.00016. ***In this work, we proposed the first version of MORGAN. The extended version of the tool is presented in Chapter 6.***

C3 - Nguyen, P.T., Di Ruscio, D., Di Rocco, J., Di Sipio, C., and Di Penta, M., *Adversarial machine learning: On the resilience of third-party library recommender systems.* In Evaluation and Assessment in Software Engineering, EASE 2021, page 247–253, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450390538. DOI: 10.1145/3463274.3463809. ***This work was presented in Chapter 9.***

C2 - Di Sipio, C., Rubei, R., Di Ruscio, D., and Nguyen, P.T., *A multinomial naïve bayesian (MNB) network to automatically recommend topics for github repositories.* In Proceedings of the Evaluation and Assessment in Software Engineering, EASE '20, page 71–80, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450377317. DOI: 10.1145/3383219.3383227. ***This work was presented in Chapter 5.***

C1 - Di Rocco, J., Di Ruscio, D., Di Sipio, C., Nguyen, P.T., and Rubei, R.. *TopFilter: An approach to recommend relevant GitHub topics.* In Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), ESEM '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375801. DOI: 10.1145/3382494.3410690. ***This work makes use of the same strategies presented in Chapter 5 to categorize GitHub repositories. However, it is not presented in any chapter of this dissertation.***

Workshops

W4 - Nguyen, P. T., Di Rocco, J., Rubei, R., Di Sipio, C., and Di Ruscio, D. (2021). Recommending Third-party Library Updates with LSTM Neural Networks. The 11th Italian Information Retrieval Workshop, 2021. URL https://ceur-ws.org/Vol-2947/paper7.pdf. ***This work presented a first version of DeepLib, an LSTM-based recommender system for migrating third-party libraries.***

W3 - Rubei, R., and Di Sipio, C. (2021). AURYGA: A Recommender System for Game Tagging. The 11th Italian Information Retrieval Workshop, 2021. URL: https://ceur-ws.org/Vol-2947/paper10.pdf. ***This work proposed an automatic approach to classify***

videogames using the same technique presented in Chapter 5. However, it is not presented in any chapter of this dissertation.

W2 - Di Rocco, J., Di Ruscio, D., Di Sipio, C., Nguyen, P. T., and Pomo, C. (2021). On the need for a body of knowledge on recommender systems. In Proceedings of the Joint KaRS and ComplexRec Workshop. URL: https://ceur-ws.org/Vol-2960/paper5.pdf. ***This work proposed a body of knowledge for RSSE. However, it is not presented in any chapter of this dissertation.***

W1 - Di Sipio, C., Di Ruscio, D., and Nguyen, P.T. *Democratizing the development of recommender systems by means of low-code platforms*, In Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, MODELS '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381352. DOI: 10.1145/3417990.3420202. ***This work proposed the foundational aspects of LEV4REC. Therefore, part of the content was presented in Chapter 8.***

Manuscripts under review/revision

M1 - Nguyen, P.T., Di Rocco J., Di Sipio, C., Rubei R., Di Ruscio D. Di Penta M. (2023). Is this Snippet Written by ChatGPT? An Empirical Study with a CodeBERT-Based Classifier, submitted to the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE 2023). ***This empirical work proposes an automated approach to detect if a code snippet is written by ChatGPT. However, it is not presented in any chapter of this dissertation and it is currently under review***

M2 - d'Aloisio, G., Di Sipio C., Di Marco A., Di Ruscio (2023), How fair are we? From conceptualization to automated assessment of fairness definitions, submitted to the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE 2023). ***This work proposed an MDE-based approach to design and test fairness assessment workflow. However, it is not presented in any chapter of this dissertation and it is currently under review***

M3 - Bergelin, J., Berardinelli, L, Bilic, D., Bruneliere, H., Cicchetti, A, Dehghani, M., Di Sipio, C, Miranda J., Rahimi, A., and Rubei, R., *Towards Automating the Design of Cyber-Physical Systems: The Experience of Volvo Construction Equipment*, Journal of Systems and Software ***This work has been developed in the frame of the Aidoart EU project and propose the application of MORGAN in an industrial context. It is currently under review.***

10.3 Future work

Analyzing source code to recommend API. Future research in this area includes *(i)* replicating the empirical evaluation on further projects, by also, possibly, supporting further programming languages, and *(ii)* updating the code base of the Eclipse Scava project[1] (which embraces all the development outcomes produced in the context of the EU CROSSMINER project) with the FOCUS tool as presented in the corresponding chapter. Furthermore, we plan to extend the user study by comparing additional approaches with FOCUS to evaluate further qualitative aspects.

Concerning LUPE, our plan is to improve the underpinning by employing additional cutting-edge techniques, such as transfer learning or attention, and by better evaluating it through user studies. Furthermore, we can make use of notable pre-trained network, i.e., CodeBERT, to improve the timing of the training phase.

Creating a taxonomy for OSS projects. We plan to improve the proposed framework further by analyzing the source code of OSS projects to compute similarity for HybridRec. Furthermore, we plan to conduct a well-structured user study by involving developers to evaluate HybridRec's outcomes. Last but not least, we will create a taxonomy of GitHub topics by grouping tags with a pairwise ranking algorithm. As a short-term plan, we plan to propose an integrated approach to generate a taxonomy of software domains by inspecting GitHub topics. In such an approach, GitHub repositories are fetched to collect different topics. Then, various filtering steps will be used to reduce the number of topics. Afterward, a reconciliation step is applied, where the selected topics are linked to Wikidata in order to help with the disambiguation of these terms. The final step is to create a discrete rank of the selected application domains by clustering algorithms. This work is currently under review.

Investigating semantic similarity to assist modelers. We plan to extend MemoRec by adding other similarity functions, e.g., structural and semantic based methods. Moreover, we can improve the encoding schemes by introducing Natural Language Pre-processing (NLP) techniques. We will augment additional information to the recommendation outcomes, e.g., type, cardinality. Afterward, we are going to conduct a proper user study with the involvement of modelers to evaluate the usability of MemoRec. Last but not least, now that we have validated the algorithmic accuracy of the proposed technique, we will integrate the conceived tool into the Eclipse IDE, providing modelers with supports embedded in their development environment.

[1] https://www.eclipse.org/scava/

We plan to further improve MORGAN by adopting different graph structures, e.g., heterogeneous or weighted graphs. Furthermore, we suppose that link prediction or generative graphs techniques can be applied as an alternative strategy to complete models represented in a graph-based format. In addition, MORGAN can be compared with existing approaches that exploit deep learning technique. Concerning the recommendation of JSON schema elements, it is our strong belief that a curated dataset with similar elements brings better results in terms of accuracy. Last but not least, we will investigate the applicability of ontologies to increase the number of relevant artifacts, moving forward a domain-aware intelligent modeling assistant capable of embedding the semantics in the retrieved recommendations.

Consolidating the gained experience in CROSSMINER. For future work, we plan to consolidate the lessons learned by applying the developed techniques and tools in other SE domains, e.g., Model-driven engineering(MDE), software testing, or Internet of Things (IoT). In particular, we already propose two different recommender system for modeling activities, i.e., MemoRec and MORGAN, that have been developed following the identified empirical guidelines. Furthermore, we will investigate the usage of cutting-edge ML-based techniques, e.g., pre-trained models, encode-decoder transformers, or large language models (LLMs).

Improving the usability of LEV4REC. At its current implementation, the platform supports the specification of configurations and the evaluation exploiting two different Python libraries. We plan to improve our conceived tool for future work by equipping it with the ability to generate code in other languages, e.g., Java, and C++. Furthermore, we will also perform more evaluations by incorporating other recommender systems and variating their initial configuration as suggested by participants of the focus group. In addition, we plan to cover the missing components, e.g., dataset generation. Moreover, we will develop a set of proper endpoints to allow RS designers to specify new features, e.g., by means of new Eclipse plugins. Last but not least, LEV4REC can be enhanced to cover features identified in the solution phase alongside the ones at the system level.

Investigating further quality aspects in RSSEs. As future work, we plan to devise and evaluate effective countermeasures that can ward off attacks tailored to RSSE. This will be done by studying learning algorithms being aware of adversarial attacks and seize them. Moreover, adversarial attempts should be turned into features for the training process, i.e., RSSE should not only learn from useful patterns, but also be able to learn how to avoid hostile patterns. Besides recommenders leveraging GitHub, we plan also to investigate to

what extent RSSE leveraging discussions from Q&A forums such as Stack Overflow (SO) are susceptible to adversarial attacks.

Our studies suggest that while the research community either underestimates or ignores it, the possibility of using falsified data to trick RSSE *is always present*, leaving a potential danger to software systems. In this respect, we see an urgent need to thoroughly perceive the likely threats to conceptualize effective defense mechanisms, thus increasing the resilience and robustness of RSSE. We consider this as part of our future research agenda. Another line of research we plan to investigate is related to fairness of RSSEs, understating to what kind of bias they are exposed. We already contributed in this direction by measuring the effect of popularity bias in TPL RSSEs. Nonetheless, there are still open challenges that needs to be addressed, e.g., evaluating additional systems or considering different types of bias.

References

[1] dex2jar. URL https://tools.kali.org/reverse-engineering/dex2jar. Library Catalog: tools.kali.org.

[2] GitHub. https://docs.github.com/en/rest/overview/resources-in-the-rest-api#rate-limiting, . Accessed: 2021-01-29.

[3] GitHub Archive Dataset. https://console.cloud.google.com/marketplace/product/github/github-repos, . Accessed: 2021-01-29.

[4] GitHub REST API v3. https://developer.github.com/v3/, . last accessed 01.12.2022.

[5] JSON schema. http://json-schema.org/. Accessed: 2022-02-29.

[6] PyGithub/PyGithub, December 2019. URL https://github.com/PyGithub/PyGithub. original-date: 2012-02-25T12:53:47Z.

[7] Understanding the search syntax - GitHub Help, 2019. URL https://help.github.com/en/github/searching-for-information-on-github/understanding-the-search-syntax.

[8] Rodin Aarssen. cwi-swat/clair: v0.1.0, September 2017. URL https://doi.org/10.5281/zenodo.891122.

[9] Shamsa Abid, Shafay Shamail, Hamid Abdul Basit, and Sarah Nadi. FACER: An API usage-based code-example recommender for opportunistic reuse. *Empirical Software Engineering*, 26(6):110, November 2021. ISSN 1382-3256, 1573-7616. doi: 10.1007/s10664-021-10000-w. URL https://link.springer.com/10.1007/s10664-021-10000-w.

[10] Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. Mining API Patterns As Partial Orders from Source Code: From Usage Scenarios to Specifications. In *6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 25–34, New York, 2007. ACM. ISBN 978-1-59593-811-4. doi: 10.1145/1287624.1287630.

[11] Lissette Almonte, Iván Cantador, Esther Guerra, and Juan de Lara. Towards automating the construction of recommender systems for low-code development platforms. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS ’20, pages 1–10, New York, NY, USA, October 2020. Association for Computing Machinery. ISBN 978-1-4503-8135-2. doi: 10.1145/3417990.3420200. URL http://doi.org/10.1145/3417990.3420200.

[12] Lissette Almonte, Esther Guerra, Iván Cantador, and Juan de Lara. Recommender systems in model-driven engineering. *Software and Systems Modeling*, July 2021. ISSN 1619-1374. doi: 10.1007/s10270-021-00905-x. URL https://doi.org/10.1007/s10270-021-00905-x.

[13] Lissette Almonte, Sara Pérez-Soler, Esther Guerra, Iván Cantador, and Juan de Lara. Automating the Synthesis of Recommender Systems for Modelling Languages. page 14, 2021.

[14] Kamel Alreshedy, Dhanush Dharmaretnam, Daniel M. German, Venkatesh Srinivasan, and T. Aaron Gulliver. SCC: Automatic classification of code snippets. *CoRR*, abs/1809.07945, 2018.

[15] Kamel Alreshedy, Dhanush Dharmaretnam, Daniel M. German, Venkatesh Srinivasan, and T. Aaron Gulliver. SCC: Automatic Classification of Code Snippets. *arXiv:1809.07945 [cs, stat]*, September 2018. URL http://arxiv.org/abs/1809.07945. arXiv: 1809.07945.

[16] Doaa Altarawy, Hossameldin Shahin, Ayat Mohammed, and Na Meng. Lascad : Language-agnostic software categorization and similar application detection. *Journal of Systems and Software*, 142, 04 2018. doi: 10.1016/j.jss.2018.04.018.

[17] Rohan Anand and Joeran Beel. Auto-surprise: An automated recommender-system (autorecsys) library with tree of parzens estimator (tpe) optimization. In *Fourteenth ACM Conference on Recommender Systems*, RecSys '20, page 585–587, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375832. doi: 10.1145/3383313.3411467. URL https://doi.org/10.1145/3383313.3411467.

[18] Vito Walter Anelli, Yashar Deldjoo, Tommaso Di Noia, Eugenio Di Sciascio, and Felice Antonio Merra. Sasha: Semantic-aware shilling attacks on recommender systems exploiting knowledge graphs. In *The Semantic Web*, pages 307–323, Cham, 2020. Springer International Publishing. ISBN 978-3-030-49461-2.

[19] Vito Walter Anelli, Alejandro Bellogin, Antonio Ferrara, Daniele Malitesta, Felice Antonio Merra, Claudio Pomo, Francesco Maria Donini, and Tommaso Di Noia. Elliot: A comprehensive and rigorous framework for reproducible recommender systems evaluation. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '21, page 2405–2414, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450380379. doi: 10.1145/3404835.3463245. URL https://doi.org/10.1145/3404835.3463245.

[20] B.J. Arnoldus, M.G.J. Brand, van den, A. Serebrenik, and J.J. Brunekreef. *Code generation with templates*. Atlantis studies in computing. Atlantis Press, Netherlands, 2012. ISBN 978-94-91216-55-8. doi: 10.2991/978-94-91216-56-5.

[21] Muhammad Hilmi Asyrofi, Ferdian Thung, David Lo, and Lingxiao Jiang. Ausearch: Accurate API usage search in github repositories with type resolution. In *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020*, pages 637–641, 2020. doi: 10.1109/SANER48275.2020.9054809. URL https://doi.org/10.1109/SANER48275.2020.9054809.

[22] Michael Backes, Sven Bugiel, and Erik Derr. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 356–367, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341394. doi: 10.1145/2976749.2978333. URL https://doi.org/10.1145/2976749.2978333.

[23] L. Bao, T. B. Le, and D. Lo. Mining sandboxes: Are we there yet? In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 445–455, March 2018. doi: 10.1109/SANER.2018.8330231.

[24] Angela Barriga, Adrian Rutle, and Rogardt Heldal. Automatic model repair using reinforcement learning. In Regina Hebig and Thorsten Berger, editors, *Proceedings of MODELS 2018 Workshops co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018), Copenhagen, Denmark, October, 14, 2018*, volume 2245 of *CEUR Workshop Proceedings*, pages 781–786. CEUR-WS.org, 2018. URL http://ceur-ws.org/Vol-2245/ammore_paper_1.pdf.

[25] B. Basten, M. Hills, P. Klint, D. Landman, A. Shahi, M. J. Steindorfer, and J. J. Vinju. M3: A General Model for Code Analytics in Rascal. In *1st International Workshop on Software Analytics*, pages 25–28, Piscataway, 2015. IEEE. doi: 10.1109/SWAN.2015.7070485.

[26] Edouard Batot and Houari Sahraoui. A generic framework for model-set selection for the unification of testing and learning MDE tasks. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, pages 374–384, Saint-malo France, October 2016. ACM. ISBN 978-1-4503-4321-3. doi: 10.1145/2976767.2976785. URL https://dl.acm.org/doi/10.1145/2976767.2976785.

[27] Pamela Baxter and Susan M. Jack. Qualitative case study methodology: Study design and implementation for novice researchers. *The Qualitative Report*, 13:544–559, 2008.

[28] Alejandro Bellogín, IváN Cantador, and Pablo Castells. A comparative study of heterogeneous item recommendations in social systems. *Inf. Sci.*, 221:142–169, February 2013. ISSN 0020-0255.

[29] Amine Benelallam, Nicolas Harrand, César Soto-Valero, Benoit Baudry, and Olivier Barais. The maven dependency graph: a temporal graph-based representation of maven central. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 344–348. IEEE, 2019.

[30] J. Bergstra, D. Yamins, and D. D. Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, page I–115–I–123. JMLR.org, 2013.

[31] Runa Bhaumik, Chad Williams, Bamshad Mobasher, and Robin Burke. Securing collaborative filtering against malicious attacks through anomaly detection. In *Proceedings of ITWP'06*, Held at AAAI 2006, Boston, Massachusetts, July 2006. URL http://www.aaai.org/Press/Reports/Workshops/ws-06-10.php.

[32] Vincent D. Blondel, Anahí Gajardo, Maureen Heymans, Pierre Senellart, and Paul Van Dooren. A measure of similarity between graph vertices: Applications to synonym extraction and web searching. *SIAM Review*, 46(4):647–666, April 2004. ISSN 0036-1445.

[33] J. Bobadilla, F. Ortega, A. Hernando, and A. Gutiérrez. Recommender systems survey. *Knowledge-Based Systems*, 46:109–132, July 2013. ISSN 09507051. doi: 10.1016/j.knosys.2013.03.012.

[34] J. Bobadilla, F. Ortega, A. Hernando, and A. Gutiérrez. Recommender systems survey. *Knowledge-Based Systems*, 46:109 – 132, 2013. ISSN 0950-7051.

[35] Hudson Borges and Marco Tulio Valente. What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform. *Journal of Systems and Software*, 146:112–129, December 2018. ISSN 01641212. doi: 10.1016/j.jss.2018.09.016. URL http://arxiv.org/abs/1811.07643. arXiv: 1811.07643.

[36] Hudson Borges, Andre Hora, and Marco Tulio Valente. Predicting the Popularity of GitHub Repositories. *Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering - PROMISE 2016*, pages 1–10, 2016. doi: 10.1145/2972958.2972966. URL http://arxiv.org/abs/1607.04342. arXiv: 1607.04342.

[37] Hudson Borges, André C. Hora, and Marco Tulio Valente. Understanding the Factors That Impact the Popularity of GitHub Repositories. In *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*, pages 334–344. IEEE Computer Society, 2016. ISBN 978-1-5090-3806-0. doi: 10.1109/ICSME.2016.31. URL https://doi.org/10.1109/ICSME.2016.31.

[38] Hudson Borges, Marco Tulio Valente, Andre Hora, and Jailton Coelho. On the Popularity of GitHub Applications: A Preliminary Note. *arXiv:1507.00604 [cs]*, March 2017. URL http://arxiv.org/abs/1507.00604. arXiv: 1507.00604.

[39] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. Model-driven software engineering in practice. *Synthesis lectures on software engineering*, 3(1):1–207, 2017.

[40] D. Breuker. Towards model-driven engineering for big data analytics – an exploratory analysis of domain-specific languages for machine learning. In *2014 47th Hawaii International Conference on System Sciences*, pages 758–767, Jan 2014. doi: 10.1109/HICSS.2014.101.

[41] Marcel Bruch, Thorsten Schäfer, and Mira Mezini. On evaluating recommender systems for API usages. In *Proceedings of the 2008 international workshop on recommendation systems for software engineering*, RSSE '08, pages 16–20, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-228-3.

[42] Loli Burgueño, Robert Clarisó, Sébastien Gérard, Shuai Li, and Jordi Cabot. An nlp-based architecture for the autocompletion of partial domain models. In Marcello La Rosa, Shazia Sadiq, and Ernest Teniente, editors, *Advanced Information Systems Engineering*, pages 91–106, Cham, 2021. Springer International Publishing. ISBN 978-3-030-79382-1.

[43] Raymond P. L. Buse and Westley Weimer. Synthesizing API Usage Examples. In *34th International Conference on Software Engineering*, pages 782–792, Piscataway, 2012. IEEE. ISBN 978-1-4673-1067-3. doi: 10.1109/ICSE.2012.6227140.

[44] X. Cai, J. Zhu, B. Shen, and Y. Chen. Greta: Graph-based tag assignment for github repositories. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 63–72, 2016. doi: 10.1109/COMPSAC.2016.124.

[45] Thibaut Capuano, Houari Sahraoui, Benoit Frenay, and Benoit Vanderose. Learning from Code Repositories to Recommend Model Classes. *The Journal of Object Technology*, 21(3):3:1, 2022. ISSN 1660-1769. doi: 10.5381/jot.2022.21.3.a4. URL http://www.jot.fm/contents/issue_2022_03/article4.html.

[46] B. Carbunar and R. Potharaju. A longitudinal study of the google app market. In *2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 242–249, 2015. doi: 10.1145/2808797.2808823.

[47] Catherine Cassell and Gillian Symon. *Essential Guide to Qualitative Methods in Organizational Research*. London, November 2022. doi: 10.4135/9781446280119.

[48] P. Castells, S. Vargas, and J. Wang. Novelty and diversity metrics for recommender systems: Choice, discovery and relevance. In *International Workshop on Diversity in Document Retrieval (DDR 2011) at the 33rd European Conference on Information Retrieval (ECIR 2011)*, Dublin, Ireland, April 2011. URL http://ir.ii.uam.es/rim3/publications/ddr11.pdf.

[49] Annie Chen. Context-Aware Collaborative Filtering System: Predicting the User's Preference in the Ubiquitous Computing Environment. In *First International Conference on Location- and Context-Awareness*, pages 244–253, Berlin, Heidelberg, 2005. Springer. ISBN 3-540-25896-5, 978-3-540-25896-4. doi: 10.1007/11426646_23.

[50] Paul-Alexandru Chirita, Wolfgang Nejdl, and Cristian Zamfir. Preventing shilling attacks in online recommender systems. In *Proceedings of the 7th Annual ACM International Workshop on Web Information and Data Management*, WIDM '05, page 67–74, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595931945. doi: 10.1145/1097047.1097061. URL https://doi.org/10.1145/1097047.1097061.

[51] Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder–decoder approaches. In *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, pages 103–111, Doha, Qatar, October 2014. Association for Computational Linguistics. doi: 10.3115/v1/W14-4012. URL https://www.aclweb.org/anthology/W14-4012.

[52] Robert Clarisó and Jordi Cabot. Applying graph kernels to model-driven engineering problems. In *Proceedings of the 1st International Workshop on Machine Learning and Software Engineering in Symbiosis*, MASES 2018, page 1–5, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450359726. doi: 10.

1145/3243127.3243128. URL https://doi-org.univaq.clas.cineca.it/10.1145/3243127.3243128.

[53] Alessandro Colantoni, Antonio Garmendia, Luca Berardinelli, Manuel Wimmer, and Johannes Bräuer. Leveraging model-driven technologies for json artefacts: The shipyard case study. In *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 250–260, 2021. doi: 10.1109/MODELS50736.2021.00033.

[54] Joel Cordeiro, Bruno Antunes, and Paulo Gomes. Context-Based Recommendation to Support Problem Solving in Software Development. In *Third International Workshop on Recommendation Systems for Software Engineering*, pages 85–89, Piscataway, 2012. IEEE. doi: 10.1109/RSSE.2012.6233418.

[55] Valerio Cosentino, Javier Luis, and Jordi Cabot. Findings from github: Methods, datasets and limitations. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, page 137–141, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341868. doi: 10.1145/2901739.2901776.

[56] Krzysztof Czarnecki. *Domain Engineering*, pages 433–444. American Cancer Society, 2002. ISBN 9780471028956. doi: 10.1002/0471028959.sof095. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/0471028959.sof095.

[57] Barthélémy Dagenais, Harold Ossher, Rachel K. E. Bellamy, Martin P. Robillard, and Jacqueline P. de Vries. Moving into a new software project landscape. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 275–284, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6. doi: 10.1145/1806799.1806842. URL http://doi.acm.org/10.1145/1806799.1806842.

[58] Jesse Davis and Mark Goadrich. The Relationship Between Precision-Recall and ROC Curves. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, pages 233–240, New York, NY, USA, 2006. ACM. ISBN 1-59593-383-2. doi: 10.1145/1143844.1143874. URL http://doi.acm.org/10.1145/1143844.1143874.

[59] Alfonso de la Vega, Diego García-Saiz, Marta Zorrilla, and Pablo Sánchez. Lavoisier: A dsl for increasing the level of abstraction of data selection and formatting in data mining. *Journal of Computer Languages*, 60:100987, 2020. ISSN 2590-1184. doi: https://doi.org/10.1016/j.cola.2020.100987. URL https://www.sciencedirect.com/science/article/pii/S2590118420300472.

[60] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, page 181–191, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357166. doi: 10.1145/3196398.3196401. URL https://doi.org/10.1145/3196398.3196401.

[61] Yashar Deldjoo, Tommaso Di Noia, and Felice Antonio Merra. Adversarial machine learning in recommender systems: State of the art and challenges. *ArXiv e-prints*,

May 2020. URL http://www-ictserv.poliba.it/publications/2020/DDM20a/Survey_AML_RecSys.pdf. Under Review.

[62] Yashar Deldjoo, Tommaso Di Noia, and Felice Antonio Merra. A survey on adversarial recommender systems: From attack/defense strategies to generative adversarial networks. *ACM Comput. Surv.*, 54(2), March 2021. ISSN 0360-0300. doi: 10.1145/3439729. URL https://doi.org/10.1145/3439729.

[63] Mukund Deshpande and George Karypis. Item-based top-<i>n</i> recommendation algorithms. *ACM Trans. Inf. Syst.*, 22(1):143–177, January 2004. ISSN 1046-8188. doi: 10.1145/963770.963776. URL https://doi.org/10.1145/963770.963776.

[64] Roberto Di Cosmo and Stefano Zacchiroli. Software Heritage: Why and How to Preserve Software Source Code. In *14th International Conference on Digital Preservation*, pages 1–10, Kyoto, 2017.

[65] Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. Mining metrics for understanding metamodel characteristics. In *Proceedings of the 6th International Workshop on Modeling in Software Engineering*, MiSE 2014, page 55–60, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328494. doi: 10.1145/2593770.2593774. URL https://doi-org.univaq.clas.cineca.it/10.1145/2593770.2593774.

[66] Juri Di Rocco, Davide Di Ruscio, Claudio Di Sipio, Phuong Nguyen, and Riccardo Rubei. Topfilter: An approach to recommend relevant github topics. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ESEM '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375801. doi: 10.1145/3382494.3410690. URL https://doi.org/10.1145/3382494.3410690.

[67] Juri Di Rocco, Davide Di Ruscio, Claudio Di Sipio, Phuong T Nguyen, and Riccardo Rubei. Development of recommendation systems for software engineering: the crossminer experience. *Empirical Software Engineering*, 26(4):1–40, 2021. URL https://doi.org/10.1007/s10664-021-09963-7.

[68] Juri Di Rocco, Claudio Di Sipio, Davide Di Ruscio, and Phuong T. Nguyen. A gnn-based recommender system to assist the specification of metamodels and models. In *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 70–81, 2021. doi: 10.1109/MODELS50736.2021.00016.

[69] Juri Di Rocco, Davide Di Ruscio, Claudio Di Sipio, Phuong T Nguyen, and Alfonso Pierantonio. Memorec: a recommender system for assisting modelers in specifying metamodels. *Software and Systems Modeling*, pages 1–21, 2022.

[70] Juri Di Rocco, Davide Di Ruscio, Claudio Di Sipio, Phuong T. Nguyen, and Riccardo Rubei. HybridRec: A recommender system for tagging GitHub repositories. *Applied Intelligence*, August 2022. ISSN 1573-7497. doi: 10.1007/s10489-022-03864-y. URL https://doi.org/10.1007/s10489-022-03864-y.

[71] Claudio Di Sipio, Davide Di Ruscio, and Phuong T. Nguyen. Democratizing the development of recommender systems by means of low-code platforms. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381352. doi: 10.1145/3417990.3420202. URL https://doi.org/10.1145/3417990.3420202.

[72] Claudio Di Sipio, Riccardo Rubei, Davide Di Ruscio, and Phuong T. Nguyen. A multinomial naïve bayesian (mnb) network to automatically recommend topics for github repositories. In *Proceedings of the Evaluation and Assessment in Software Engineering*, EASE '20, page 71–80, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450377317. doi: 10.1145/3383219.3383227. URL https://doi.org/10.1145/3383219.3383227.

[73] Claudio Di Sipio, Juri Di Rocco, Davide Di Ruscio, and Phuong Thanh Nguyen. A low-code tool supporting the development of recommender systems. In *Fifteenth ACM Conference on Recommender Systems*, RecSys '21, page 741–744, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384582. doi: 10.1145/3460231.3478885. URL https://doi.org/10.1145/3460231.3478885.

[74] Qiming Diao, Minghui Qiu, Chao-Yuan Wu, Alexander J. Smola, Jing Jiang, and Chong Wang. Jointly modeling aspects, ratings and sentiments for movie recommendation (jmars). In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, page 193–202, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450329569. doi: 10.1145/2623330.2623758. URL https://doi.org/10.1145/2623330.2623758.

[75] Davide Di Ruscio, Dimitris Kolovos, Juan de Lara, Alfonso Pierantonio, Massimo Tisi, and Manuel Wimmer. Low-code development and model-driven engineering: Two sides of the same coin? *Software and Systems Modeling*, 21(2):437–446, April 2022. ISSN 1619-1374. doi: 10.1007/s10270-021-00970-2. URL https://doi.org/10.1007/s10270-021-00970-2.

[76] Claudio Di Sipio, Juri Di Rocco, Davide Di Ruscio, and Phuong T. Nguyen. MORGAN: a modeling recommender system based on graph kernel. *Software and Systems Modeling*, April 2023. ISSN 1619-1374. doi: 10.1007/s10270-023-01102-8. URL https://doi.org/10.1007/s10270-023-01102-8.

[77] G. Dupont, S. Mustafiz, F. Khendek, and M. Toeroe. Building Domain-Specific Modelling Environments with Papyrus: An Experience Report. In *2018 IEEE/ACM 10th International Workshop on Modelling in Software Engineering (MiSE)*, pages 49–56, May 2018. ISSN: 2575-4475.

[78] Holly Edmunds. The Focus Group Research Handbook. *The Bottom Line*, 12(3): 46–46, January 1999. ISSN 0888-045X. doi: 10.1108/bl.1999.12.3.46.1. URL https://doi.org/10.1108/bl.1999.12.3.46.1. Publisher: Emerald Group Publishing Limited.

[79] Michael D. Ekstrand. LensKit for Python: Next-Generation Software for Recommender Systems Experiments. In *Proceedings of the 29th ACM International Confer-*

ence on Information & Knowledge Management, pages 2999–3006, Virtual Event Ireland, October 2020. ACM. ISBN 978-1-4503-6859-9. doi: 10.1145/3340531.3412778.

[80] Chun-I Fan, Han-Wei Hsiao, Chun-Han Chou, and Yi-Fan Tseng. Malware detection systems based on api log data mining. In *Proceedings of the 2015 IEEE 39th Annual Computer Software and Applications Conference - Volume 03*, COMPSAC '15, page 255–260, USA, 2015. IEEE Computer Society. ISBN 9781467365642. doi: 10.1109/COMPSAC.2015.241. URL https://doi.org/10.1109/COMPSAC.2015.241.

[81] Mattia Fazzini, Qi Xin, and Alessandro Orso. Automated api-usage update for android apps. In *Proceedings of the 28th ISSTA*, ISSTA 2019, page 204–215, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362245. doi: 10.1145/3293882.3330571.

[82] Alexander Felfernig, Viet-Man Le, Andrei Popescu, Mathias Uta, Thi Ngoc Trang Tran, and Müslüm Atas. An overview of recommender systems and machine learning in feature modeling and configuration. In *15th International Working Conference on Variability Modelling of Software-Intensive Systems*, VaMoS'21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450388245. doi: 10.1145/3442391.3442408. URL https://doi-org.univaq.clas.cineca.it/10.1145/3442391.3442408.

[83] Jaroslav Fowkes and Charles Sutton. Parameter-free Probabilistic API Mining Across GitHub. In *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 254–265, New York, 2016. ACM. ISBN 978-1-4503-4218-6. doi: 10.1145/2950290.2950319.

[84] Kavita Ganesan. Topic Suggestions for Millions of Repositories - The GitHub Blog, 2017. URL https://github.blog/2017-07-31-topics/.

[85] Zeno Gantner, Steffen Rendle, Christoph Freudenthaler, and Lars Schmidt-Thieme. Mymedialite: A free recommender system library. In *Proceedings of the Fifth ACM Conference on Recommender Systems*, RecSys '11, page 305–308, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450306836. doi: 10.1145/2043932.2043989. URL https://doi-org.univaq.clas.cineca.it/10.1145/2043932.2043989.

[86] Joshua Garcia, Mahmoud Hammad, and Sam Malek. Lightweight, Obfuscation-Resilient Detection and Family Identification of Android Malware. *ACM Transactions on Software Engineering and Methodology*, 26(3):11:1–11:29, January 2018. ISSN 1049-331X. doi: 10.1145/3162625. URL http://doi.org/10.1145/3162625.

[87] Vicente García-Díaz, Jordán Pascual Espada, Begoña Cristina Pelayo García Bustelo, and Juan Manuel Cueva Lovelle. Towards a standard-based domain-specific platform to solve machine learning-based problems. *IJIMAI*, 3(5):6–12, 2015.

[88] Pankaj K. Garg, Shinji Kawaguchi, Makoto Matsushita, and Katsuro Inoue. MUDABlue: An automatic categorization system for open source repositories. *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, pages 184–193, 2004. ISSN 1530-1362.

[89] Marko Gasparic, Andrea Janes, Francesco Ricci, Gail C. Murphy, and Tural Gurbanov. A graphical user interface for presenting integrated development environment command recommendations: Design, evaluation, and implementation. *Information and Software Technology*, 92:236–255, 2017. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2017.08.006. URL https://www.sciencedirect.com/science/article/pii/S0950584916303524.

[90] Mouzhi Ge, Carla Delgado-Battenfeld, and Dietmar Jannach. Beyond accuracy: Evaluating recommender systems by coverage and serendipity. In *Proceedings of the Fourth ACM Conference on Recommender Systems*, RecSys '10, page 257–260, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605589060. doi: 10.1145/1864708.1864761. URL https://doi.org/10.1145/1864708.1864761.

[91] F. Geiger, I. Malavolta, L. Pascarella, F. Palomba, D. Di Nucci, and A. Bacchelli. A graph-based dataset of commit history of real-world android apps. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 30–33, 2018.

[92] Sanjoy Ghose and Oded Lowengart. Taste tests: Impacts of consumer perceptions and preferences on brand positioning strategies. *Journal of Targeting, Measurement and Analysis for Marketing*, 10(1):26–41, August 2001. ISSN 1479-1862.

[93] Heather J. Goldsby and Betty H.C. Cheng. Avida-MDE: a digital evolution approach to generating models of adaptive software behavior. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation - GECCO '08*, page 1751, Atlanta, GA, USA, 2008. ACM Press. ISBN 978-1-60558-130-9. doi: 10.1145/1389095.1389434. URL http://portal.acm.org/citation.cfm?doid=1389095.1389434.

[94] Qingyuan Gong, Jiayun Zhang, Yang Chen, Qi Li, Yu Xiao, Xin Wang, and Pan Hui. Detecting malicious accounts in online developer communities using deep learning. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, CIKM '19, page 1251–1260, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450369763. doi: 10.1145/3357384.3357971. URL https://doi.org/10.1145/3357384.3357971.

[95] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In Yoshua Bengio and Yann LeCun, editors, *ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1412.6572.

[96] Georgios Gousios. The ghtorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-2936-1. URL http://dl.acm.org/citation.cfm?id=2487085.2487132.

[97] Georgios Gousios and Diomidis Spinellis. Ghtorrent: Github's data from a firehose. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 12–21. IEEE, 2012.

[98] Edouard Grave, Piotr Bojanowski, Prakhar Gupta, Armand Joulin, and Tomas Mikolov. Learning word vectors for 157 languages. In *Proceedings of the Eleventh International*

Conference on Language Resources and Evaluation (LREC 2018), Miyazaki, Japan, May 2018. European Language Resources Association (ELRA). URL https://www.aclweb.org/anthology/L18-1550.

[99] Robert J. Grissom and John J. Kim. *Effect sizes for research: A broad practical approach*. Lawrence Earlbaum Associates, 2nd edition edition, 2005.

[100] Richard Gronback. Eclipse Modeling Project | The Eclipse Foundation, March 2021. URL https://www.eclipse.org/modeling/emf/.

[101] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep API Learning. In *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 631–642, New York, 2016. ACM. ISBN 978-1-4503-4218-6. doi: 10.1145/2950290.2950334.

[102] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep Code Search. In *40th International Conference on Software Engineering*, pages 933–944, New York, 2018. ACM. ISBN 978-1-4503-5638-1. doi: 10.1145/3180155.3180167.

[103] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Codekernel: A graph kernel based approach to the selection of API usage examples. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pages 590–601, 2019. doi: 10.1109/ASE.2019.00061. URL https://doi.org/10.1109/ASE.2019.00061.

[104] Ihsan Gunes, Cihan Kaleli, Alper Bilge, and Huseyin Polat. Shilling attacks against recommender systems: A comprehensive survey. *Artif. Intell. Rev.*, 42(4):767–799, December 2014. ISSN 0269-2821. doi: 10.1007/s10462-012-9364-9. URL https://doi.org/10.1007/s10462-012-9364-9.

[105] F. Maxwell Harper and Joseph A. Konstan. The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, 5(4), December 2015. ISSN 2160-6455. doi: 10.1145/2827872. URL https://doi.org/10.1145/2827872.

[106] Q. He, B. Li, F. Chen, J. Grundy, X. Xia, and Y. Yang. Diversified third-party library prediction for mobile app development. *IEEE Transactions on Software Engineering*, pages 1–1, 2020.

[107] Jonathan L. Herlocker, Joseph A. Konstan, Loren G. Terveen, and John T. Riedl. Evaluating collaborative filtering recommender systems. *ACM Trans. Inf. Syst.*, 22 (1):5–53, jan 2004. ISSN 1046-8188. doi: 10.1145/963770.963772. URL https://doi.org/10.1145/963770.963772.

[108] Mark Hills and Paul Klint. Php air: Analyzing php systems with rascal. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 454–457. IEEE, 2014.

[109] Jerry L. Hintze and Ray D. Nelson. Violin plots: A box plot-density trace synergism. *The American Statistician*, 52(2):181–184, 1998. doi: 10.1080/00031305.1998.10480559. URL https://amstat.tandfonline.com/doi/abs/10.1080/00031305.1998.10480559.

[110] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL https://doi.org/10.1162/neco.1997.9.8.1735.

[111] Reid Holmes and Gail C. Murphy. Using Structural Context to Recommend Source Code Examples. In *27th International Conference on Software Engineering*, pages 117–125, New York, 2005. ACM. ISBN 1-58113-963-2. doi: 10.1145/1062455.1062491.

[112] Reid Holmes, Robert J. Walker, and Gail C. Murphy. Strathcona example recommendation tool. In Michel Wermelinger and Harald C. Gall, editors, *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 237–240. ACM, 2005. ISBN 1-59593-014-0. doi: 10.1145/1081706.1081744. URL https://doi.org/10.1145/1081706.1081744.

[113] Jeremy Howard and Sylvain Gugger. fastai: A Layered API for Deep Learning. *Information*, 11(2):108, February 2020. ISSN 2078-2489. doi: 10.3390/info11020108. URL http://arxiv.org/abs/2002.04688. arXiv: 2002.04688.

[114] Ling Huang, Anthony D. Joseph, Blaine Nelson, Benjamin I.P. Rubinstein, and J. D. Tygar. Adversarial machine learning. In *Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence*, AISec '11, page 43–58, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450310031. doi: 10.1145/2046684.2046692. URL https://doi.org/10.1145/2046684.2046692.

[115] Yujin Huang, Han Hu, and Chunyang Chen. Robustness of on-device models: Adversarial attack to deep learning models on android apps. In *Proceedings of the 43rd International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '21, page 101–110. IEEE Press, 2021. ISBN 9780738146690. doi: 10.1109/ICSE-SEIP52600.2021.00019. URL https://doi.org/10.1109/ICSE-SEIP52600.2021.00019.

[116] Nicolas Hug. Surprise: A Python library for recommender systems. *Journal of Open Source Software*, 5(52):2174, August 2020. ISSN 2475-9066. doi: 10.21105/joss.02174. URL https://joss.theoj.org/papers/10.21105/joss.02174.

[117] John Hutchinson, Jon Whittle, and Mark Rouncefield. Model-driven engineering practices in industry. *Sci. Comput. Program.*, 89(PB):144–161, sep 2014. ISSN 0167-6423. doi: 10.1016/j.scico.2013.03.017. URL https://doi.org/10.1016/j.scico.2013.03.017.

[118] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren, editors. *Automated Machine Learning: Methods, Systems, Challenges*. The Springer Series on Challenges in Machine Learning. Springer International Publishing, Cham, 2019. ISBN 978-3-030-05317-8 978-3-030-05318-5. doi: 10.1007/978-3-030-05318-5. URL http://link.springer.com/10.1007/978-3-030-05318-5.

[119] Felicien Ihirwe, Davide Di Ruscio, Silvia Mazzini, Pierluigi Pierini, and Alfonso Pierantonio. Low-code engineering for internet of things: A state of research. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS '20, New York,

NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381352. doi: 10.1145/3417990.3420208. URL https://doi.org/10.1145/3417990.3420208.

[120] Maliheh Izadi, Abbas Heydarnoori, and Georgios Gousios. Topic recommendation for software repositories using multi-label classification algorithms. *Empirical Software Engineering*, 26(5):93, July 2021. ISSN 1573-7616. doi: 10.1007/s10664-021-09976-2. URL https://doi.org/10.1007/s10664-021-09976-2.

[121] Jing Jiang, David Lo, Jiahuan He, Xin Xia, Pavneet Singh Kochhar, and Li Zhang. Why and How Developers Fork What from Whom in GitHub. *Empirical Software Engineering*, 22(1):547–578, February 2017. ISSN 1382-3256. doi: 10.1007/s10664-016-9436-6. URL https://doi.org/10.1007/s10664-016-9436-6.

[122] Liangxiao Jiang, Dianhong Wang, Zhihua Cai, and Xuesong Yan. Survey of improving naive bayes for classification. In Reda Alhajj, Hong Gao, Jianzhong Li, Xue Li, and Osmar R. Zaïane, editors, *Advanced Data Mining and Applications*, pages 134–145, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-73871-8.

[123] Jose Luis Jorro-Aragoneses, Belén Díaz-Agudo, Juan A. Recio-García, and Guillermo Jimenez-Díaz. RecoLibry Suite: a set of intelligent tools for the development of recommender systems. *Automated Software Engineering*, 27(1-2):63–89, June 2020. ISSN 0928-8910, 1573-7535. doi: 10.1007/s10515-020-00269-4. URL http://link.springer.com/10.1007/s10515-020-00269-4.

[124] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. Germán, and Daniela E. Damian. The promises and perils of mining github. In *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*, pages 92–101, 2014.

[125] George Karypis. Evaluation of item-based top-n recommendation algorithms. In *Procs. of the tenth international conf. on information and knowledge management*, CIKM ’01, pages 247–254, New York, NY, USA, 2001. ACM. ISBN 1-58113-436-3.

[126] Steven Kelly and Juha-Pekka Tolvanen. Domain-specific modeling - enabling full code generation. 2008.

[127] M. G. Kendall. A New Measure of Rank Correlation. *Biometrika*, 30(1/2):81–93, 1938. ISSN 00063444. URL http://www.jstor.org/stable/2332226.

[128] Ashraf M. Kibriya, Eibe Frank, Bernhard Pfahringer, and Geoffrey Holmes. Multinomial naive bayes for text categorization revisited. In Geoffrey I. Webb and Xinghuo Yu, editors, *AI 2004: Advances in Artificial Intelligence*, pages 488–499, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-30549-1.

[129] Barbara A. Kitchenham, Pearl Brereton, Zhi Li, David Budgen, and Andrew James Burn. Repeatability of systematic literature reviews. In *15th International Conference on Evaluation & Assessment in Software Engineering, EASE 2011, Durham, UK, 11-12 April 2011, Proceedings*, pages 46–55, 2011. doi: 10.1049/ic.2011.0006. URL https://doi.org/10.1049/ic.2011.0006.

[130] R. Koch. *The 80/20 Principle: The Secret of Achieving More with Less*. A Currency book. Doubleday, 1999. ISBN 9780385491747.

[131] William Koch, Abdelberi Chaabane, Manuel Egele, William Robertson, and Engin Kirda. Semi-automated discovery of server-based information oversharing vulnerabilities in Android applications. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 147–157, New York, NY, USA, July 2017. Association for Computing Machinery. ISBN 978-1-4503-5076-1. doi: 10.1145/3092703.3092708. URL http://doi.org/10.1145/3092703.3092708.

[132] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, pages 1137–1143, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc. ISBN 1-55860-363-8. URL http://dl.acm.org/citation.cfm?id=1643031.1643047.

[133] J. Kontio, L. Lehtola, and J. Bragge. Using the focus group method in software engineering: obtaining practitioner and user experiences. In *Proceedings. 2004 International Symposium on Empirical Software Engineering, 2004. ISESE '04.*, pages 271–280, 2004. doi: 10.1109/ISESE.2004.1334914.

[134] Arseny Korotaev and Lyudmila Lyadova. Method for the Development of Recommendation Systems, Customizable to Domains, with Deep GRU Network:. In *Proceedings of the 10th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management*, pages 231–236, Seville, Spain, 2018. SCITEPRESS - Science and Technology Publications. ISBN 978-989-758-330-8. doi: 10.5220/0006933302310236. URL http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0006933302310236.

[135] Lars Kotthoff, Chris Thornton, Holger H. Hoos, Frank Hutter, and Kevin Leyton-Brown. Auto-weka 2.0: Automatic model selection and hyperparameter optimization in weka. *Journal of Machine Learning Research*, 18(25):1–5, 2017. URL http://jmlr.org/papers/v18/16-261.html.

[136] Nils M. Kriege, Pierre-Louis Giscard, and Richard Wilson. On Valid Optimal Assignment Kernels and Applications to Graph Classification. In *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016. URL https://proceedings.neurips.cc/paper/2016/hash/0efe32849d230d7f53049ddc4a4b0c60-Abstract.html.

[137] Tobias Kuschke, Patrick Mäder, and Patrick Rempel. Recommending Auto-completions for Software Modeling Activities. In Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter Clarke, editors, *Model-Driven Engineering Languages and Systems*, Lecture Notes in Computer Science, pages 170–186, Berlin, Heidelberg, 2013. Springer. ISBN 978-3-642-41533-3. doi: 10.1007/978-3-642-41533-3_11. 00015.

[138] LASER and LASER. *Software engineering: international summer schools, LASER 2013-2014, Elba, Italy: revised tutorial lectures*. Number 8987 in Lecture notes in computer science Programming and software engineering. Springer, [Cham] Heidelberg, 2015. ISBN 978-3-319-28405-7 978-3-319-28406-4.

[139] Sungho Lee and Sukyoung Ryu. Adlib: analyzer for mobile ad platform libraries. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software*

Testing and Analysis, ISSTA 2019, pages 262–272, New York, NY, USA, July 2019. Association for Computing Machinery. ISBN 978-1-4503-6224-5. doi: 10.1145/3293882.3330562. URL http://doi.org/10.1145/3293882.3330562.

[140] VI Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, 1966.

[141] VI Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, 1966.

[142] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, and W. Huo. Libd: Scalable and precise third-party library detection in android markets. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 335–346, 2017.

[143] Xiang Li, Huaimin Wang, Gang Yin, Tao Wang, Cheng Yang, Yue Yu, and Dengqing Tang. Inducing Taxonomy from Tags: An Agglomerative Hierarchical Clustering Framework. In Shuigeng Zhou, Songmao Zhang, and George Karypis, editors, *Advanced Data Mining and Applications*, pages 64–77, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-35527-1.

[144] Mario Linares-Vásquez, Collin Mcmillan, Denys Poshyvanyk, and Mark Grechanik. On using machine learning to automatically classify software applications into domain categories. *Empirical Softw. Engg.*, 19(3):582–618, June 2014. ISSN 1382-3256. doi: 10.1007/s10664-012-9230-z.

[145] Chunyang Ling, Yanzhen Zou, and Bing Xie. Graph neural network based collaborative filtering for api usage recommendation. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 36–47, 2021. doi: 10.1109/SANER50967.2021.00013.

[146] Francesca Lonetti, Vânia de Oliveira Neves, and Antonia Bertolino. Designing and testing systems of systems: From variability models to test cases passing through desirability assessment. *Journal of Software: Evolution and Process*, 34(10), October 2022. ISSN 2047-7473, 2047-7481. doi: 10.1002/smr.2427. URL https://onlinelibrary.wiley.com/doi/10.1002/smr.2427.

[147] José Antonio Hernández López and Jesús Sánchez Cuadrado. Mar: A structure-based search engine for models. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, MODELS '20, page 57–67, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370196. doi: 10.1145/3365438.3410947. URL https://doi.org/10.1145/3365438.3410947.

[148] José Antonio Hernández López and Jesús Sánchez Cuadrado. Mar: a structure-based search engine for models. In *Proceedings of the 23rd ACM/IEEE international conference on model driven engineering languages and systems*, pages 57–67, 2020.

[149] José Antonio Hernández López, Javier Luis Cánovas Izquierdo, and Jesús Sánchez Cuadrado. Modelset: a dataset for machine learning in model-driven engineering. *Software and Systems Modeling*, 21(3):967–986, 2022.

[150] Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. In Lluís Màrquez, Chris Callison-Burch, Jian Su, Daniele Pighin, and Yuval Marton, editors, *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015, Lisbon, Portugal, September 17-21, 2015*, pages 1412–1421. The Association for Computational Linguistics, 2015. doi: 10.18653/v1/d15-1166. URL https://doi.org/10.18653/v1/d15-1166.

[151] Fei Lv, Hongyu Zhang, Jian-Guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. CodeHow: Effective code search based on API understanding and extended boolean model (E). In *30th IEEE/ACM international conference on automated software engineering, ASE 2015, lincoln, NE, USA, november 9-13, 2015*, pages 260–270, 2015.

[152] Jesús J. López-Fernández, Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. Example-driven meta-model development. *Software & Systems Modeling*, 14(4):1323–1347, October 2015. ISSN 1619-1366, 1619-1374. doi: 10.1007/s10270-013-0392-y. URL http://link.springer.com/10.1007/s10270-013-0392-y.

[153] Stephen G. MacDonell, Martin J. Shepperd, Barbara A. Kitchenham, and Emilia Mendes. How reliable are systematic reviews in empirical software engineering? *IEEE Trans. Software Eng.*, 36(5):676–687, 2010. doi: 10.1109/TSE.2010.28. URL https://doi.org/10.1109/TSE.2010.28.

[154] Arun S. Maiya. ktrain: A Low-Code Library for Augmented Machine Learning. *arXiv:2004.10703 [cs]*, July 2020. URL http://arxiv.org/abs/2004.10703. arXiv: 2004.10703.

[155] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, USA, 2008. ISBN 0521865719.

[156] Wes McKinney et al. pandas: a foundational python library for data analysis and statistics. *Python for High Performance and Scientific Computing*, 14(9):1–9, 2011.

[157] Collin McMillan, Denys Poshyvanyk, and Mark Grechanik. Recommending source code examples via API call usages and documentation. In *Proceedings of the 2Nd international workshop on recommendation systems for software engineering*, RSSE '10, pages 21–25, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-974-9.

[158] Collin McMillan, Mark Grechanik, and Denys Poshyvanyk. Detecting similar software applications. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 364–374, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1067-3.

[159] Hector Mendoza, Aaron Klein, Matthias Feurer, Jost Tobias Springenberg, Matthias Urban, Michael Burkart, Maximilian Dippel, Marius Lindauer, and Frank Hutter. *Towards Automatically-Tuned Deep Neural Networks*, pages 135–149. Springer International Publishing, Cham, 2019. ISBN 978-3-030-05318-5. doi: 10.1007/978-3-030-05318-5_7. URL https://doi.org/10.1007/978-3-030-05318-5_7.

[160] Kim Mens and Angela Lozano. Source code-based recommendation systems. In Martin P. Robillard, Walid Maalej, Robert J. Walker, and Thomas Zimmermann, editors, *Recommendation Systems in Software Engineering*, pages 93–130. Springer, 2014. doi: 10.1007/978-3-642-45135-5_5. URL https://doi.org/10.1007/978-3-642-45135-5_5.

[161] Christos Mettouris, Achilleas Achilleos, Georgia Kapitsaki, and George A. Papadopoulos. The UbiCARS Model-Driven Framework: Automating Development of Recommender Systems for Commerce. In Achilles Kameas and Kostas Stathis, editors, *Ambient Intelligence*, Lecture Notes in Computer Science, pages 37–53, Cham, 2018. Springer International Publishing. ISBN 978-3-030-03062-9. doi: 10.1007/978-3-030-03062-9_3.

[162] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, page 3111–3119, Red Hook, NY, USA, 2013. Curran Associates Inc.

[163] George A. Miller. Wordnet: A lexical database for english. *Commun. ACM*, 38 (11):39–41, November 1995. ISSN 0001-0782. doi: 10.1145/219717.219748. URL http://doi.acm.org/10.1145/219717.219748.

[164] B. Mobasher, R. Burke, R. Bhaumik, and J. J. Sandvig. Attacks and remedies in collaborative recommendation. *IEEE Intelligent Systems*, 22(3):56–63, 2007.

[165] Piero Molino, Yaroslav Dudin, and Sai Sumanth Miryala. Ludwig: a type-based declarative deep learning toolbox. *arXiv:1909.07930 [cs, stat]*, September 2019. URL http://arxiv.org/abs/1909.07930. arXiv: 1909.07930.

[166] Àngel Mora Segura and Juan de Lara. Extremo: An Eclipse plugin for modelling and meta-modelling assistance. *Science of Computer Programming*, 180:71–80, 2019. ISSN 0167-6423. doi: 10.1016/j.scico.2019.05.003. URL https://www.sciencedirect.com/science/article/pii/S0167642319300644.

[167] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. How can I use this method? In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 880–890, 2015.

[168] Gail C. Murphy. Attacking information overload in software development. In *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2009, Corvallis, OR, USA, 20-24 September 2009, Proceedings*, page 4, 2009.

[169] Emerson R. Murphy-Hill, Gail C. Murphy, and William G. Griswold. Understanding context: creating a lasting impact in experimental software engineering research. In *Proceedings of FoSER 2010, at FSE 2010, Santa Fe, NM, USA, November 7-11, 2010*, pages 255–258, 2010.

[170] Gunter Mussbacher, Benoit Combemale, Silvia Abrahão, Nelly Bencomo, Loli Burgueño, Gregor Engels, Jörg Kienzle, Thomas Kühn, Sébastien Mosser, Houari

Sahraoui, and Martin Weyssow. Towards an assessment grid for intelligent modeling assistance. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381352. doi: 10.1145/3417990.3421396. URL https://doi-org.univaq.clas.cineca.it/10.1145/3417990.3421396.

[171] Gunter Mussbacher, Benoit Combemale, Jörg Kienzle, Silvia Abrahão, Hyacinth Ali, Nelly Bencomo, Márton Búr, Loli Burgueño, Gregor Engels, Pierre Jeanjean, Jean-Marc Jézéquel, Thomas Kühn, Sébastien Mosser, Houari Sahraoui, Eugene Syriani, Dániel Varró, and Martin Weyssow. Opportunities in intelligent modeling assistance. *Software and Systems Modeling*, 19(5):1045–1053, September 2020. ISSN 1619-1366, 1619-1374. doi: 10.1007/s10270-020-00814-5. URL http://link.springer.com/10.1007/s10270-020-00814-5.

[172] Annamalai Narayanan, Mahinthan Chandramohan, Lihui Chen, and Yang Liu. A multi-view context-aware approach to Android malware detection and malicious code localization. *Empirical Software Engineering*, 23(3):1222–1274, June 2018. ISSN 1573-7616. doi: 10.1007/s10664-017-9539-8. URL https://doi.org/10.1007/s10664-017-9539-8.

[173] Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. What Makes a Good Code Example?: A Study of Programming Q&A in StackOverflow. In *28th IEEE International Conference on Software Maintenance*, pages 25–34, Piscataway, 2012. IEEE. doi: 10.1109/ICSM.2012.6405249.

[174] Anh Mai Nguyen, Jason Yosinski, and Jeff Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *CVPR*, pages 427–436. IEEE Computer Society, 2015. ISBN 978-1-4673-6964-0. URL http://dblp.uni-trier.de/db/conf/cvpr/cvpr2015.html#NguyenYC15.

[175] P. T. Nguyen, J. Di Rocco, R. Rubei, and D. Di Ruscio. CrossSim: Exploiting mutual relationships to detect similar OSS projects. In *2018 44th euromicro conference on software engineering and advanced applications (SEAA)*, pages 388–395, August 2018.

[176] P. T. Nguyen, J. Di Rocco, D. Di Ruscio, A. Pierantonio, and L. Iovino. Automated classification of metamodel repositories: A machine learning approach. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 272–282, Sep. 2019. doi: 10.1109/MODELS.2019.00011.

[177] P. T. Nguyen, J. Di Rocco, C. Di Sipio, D. Di Ruscio, and M. Di Penta. Recommending api function calls and code snippets to support software development. *IEEE Transactions on Software Engineering*, pages 1–1, 2021. doi: 10.1109/TSE.2021.3059907.

[178] Phuong Nguyen, Paolo Tomeo, Tommaso Di Noia, and Eugenio Di Sciascio. An evaluation of simrank and personalized pagerank to build a recommender system for the web of data. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15 Companion, page 1477–1482, New York, NY, USA, 2015.

Association for Computing Machinery. ISBN 9781450334730. doi: 10.1145/2740908.2742141. URL https://doi.org/10.1145/2740908.2742141.

[179] Phuong T. Nguyen, Paolo Tomeo, Tommaso Di Noia, and Eugenio Di Sciascio. Content-based recommendations via DBpedia and freebase: A case study in the music domain. In *Proceedings of the 14th international conference on the semantic web - ISWC 2015 - volume 9366*, pages 605–621, New York, NY, USA, 2015. Springer-Verlag New York, Inc. ISBN 978-3-319-25006-9.

[180] Phuong T. Nguyen, Juri Di Rocco, and Davide Di Ruscio. Mining software repositories to support OSS developers: A recommender systems approach. In *Proceedings of the 9th italian information retrieval workshop, rome, italy, may, 28-30, 2018.*, 2018.

[181] Phuong T. Nguyen, Juri Di Rocco, Davide Di Ruscio, Lina Ochoa, Thomas Degueule, and Massimiliano Di Penta. FOCUS: A Recommender System for Mining API Function Calls and Usage Patterns. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 1050–1060, Piscataway, NJ, USA, 2019. IEEE Press. doi: 10.1109/ICSE.2019.00109. URL https://doi.org/10.1109/ICSE.2019.00109.

[182] Phuong T. Nguyen, Juri Di Rocco, Davide Di Ruscio, and Massimiliano Di Penta. Crossrec: Supporting software developers by recommending third-party libraries. *Journal of Systems and Software*, 161:110460, 2020. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2019.110460. URL https://www.sciencedirect.com/science/article/pii/S0164121219302341.

[183] Phuong T. Nguyen, Juri Di Rocco, Riccardo Rubei, and Davide Di Ruscio. An automated approach to assess the similarity of GitHub repositories. *Softw. Qual. J.*, 28 (2):595–631, 2020. doi: 10.1007/s11219-019-09483-0. URL https://doi.org/10.1007/s11219-019-09483-0.

[184] Phuong T. Nguyen, Juri Di Rocco, Riccardo Rubei, and Davide Di Ruscio. An automated approach to assess the similarity of GitHub repositories. *Software Quality Journal*, feb 2020. doi: 10.1007/s11219-019-09483-0. URL https://doi.org/10.1007%2Fs11219-019-09483-0.

[185] Phuong T. Nguyen, Davide Di Ruscio, Alfonso Pierantonio, Juri Di Rocco, and Ludovico Iovino. Convolutional neural networks for enhanced classification mechanisms of metamodels. *Journal of Systems and Software*, page 110860, 2020. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2020.110860. URL http://www.sciencedirect.com/science/article/pii/S0164121220302508.

[186] Phuong T. Nguyen, Juri Di Rocco, Claudio Di Sipio, Davide Di Ruscio, and Massimiliano Di Penta. TSE FOCUS replication package, January 2021. URL https://doi.org/10.5281/zenodo.4415618.

[187] Phuong T. Nguyen, Davide Di Ruscio, Juri Di Rocco, Claudio Di Sipio, and Massimiliano Di Penta. Adversarial machine learning: On the resilience of third-party library recommender systems. In *Evaluation and Assessment in Software Engineering*,

EASE 2021, page 247–253, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450390538. doi: 10.1145/3463274.3463809. URL https://doi.org/10.1145/3463274.3463809.

[188] Phuong T. Nguyen, Claudio Di Sipio, Juri Di Rocco, Massimiliano Di Penta, and Davide Di Ruscio. Adversarial attacks to api recommender systems: Time to wake up and smell the coffee? In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 253–265, 2021. doi: 10.1109/ASE51524.2021.9678946.

[189] Phuong T. Nguyen, Claudio Di Sipio, Juri Di Rocco, Davide Di Ruscio, and Massimiliano Di Penta. APIRecSys-AML: Artifact Evaluation, 2021. URL https://doi.org/10.5281/zenodo.5105955.

[190] Phuong T. Nguyen, Claudio Di Sipio, Juri Di Rocco, Davide Di Ruscio, and Massimiliano Di Penta. Fitting missing api puzzles with machine translation techniques. *Expert Systems with Applications*, 216:119477, 2023. ISSN 0957-4174. doi: https://doi.org/10.1016/j.eswa.2022.119477. URL https://www.sciencedirect.com/science/article/pii/S0957417422024964.

[191] Tam The Nguyen, Hung Viet Pham, Phong Minh Vu, and Tung Thanh Nguyen. Learning API usages from bytecode: a statistical approach. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 416–427, 2016. doi: 10.1145/2884781.2884873. URL https://doi.org/10.1145/2884781.2884873.

[192] Haoran Niu, Iman Keivanloo, and Ying Zou. API Usage Pattern Recommendation for Software Development. *Journal of Systems and Software*, 129(C):127–139, 2017. ISSN 0164-1212. doi: 10.1016/j.jss.2016.07.026.

[193] Haoran Niu, Iman Keivanloo, and Ying Zou. API usage pattern recommendation for software development. *Journal of Systems and Software*, 129:127–139, July 2017. ISSN 01641212.

[194] Tommaso Di Noia and Vito Claudio Ostuni. Recommender systems and linked open data. In Wolfgang Faber and Adrian Paschke, editors, *Reasoning Web. Web Logic Rules - 11th International Summer School 2015, Berlin, Germany, July 31 - August 4, 2015, Tutorial Lectures*, volume 9203 of *Lecture Notes in Computer Science*, pages 88–113. Springer, 2015. doi: 10.1007/978-3-319-21768-0_4. URL https://doi.org/10.1007/978-3-319-21768-0_4.

[195] Christopher Olah. Understanding LSTM Networks, May 2020. URL https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[196] A. N. Oppenheim. *Questionnaire Design, Interviewing and Attitude Measurement*. Pinter Publishers, 1992.

[197] Ali Ouni, Raula Gaikovina Kula, Marouane Kessentini, Takashi Ishio, Daniel M. German, and Katsuro Inoue. Search-based software library recommendation using multi-objective optimization. *Inf. Softw. Technol.*, 83(C):55–75, March 2017. ISSN 0950-5849. doi: 10.1016/j.infsof.2016.11.007. URL https://doi.org/10.1016/j.infsof.2016.11.007.

[198] S. Pantelimon, T. Rogojanu, A. Braileanu, V. Stanciu, and C. Dobre. Towards a seamless integration of iot devices with iot platforms using a low-code approach. In *2019 IEEE 5th World Forum on Internet of Things (WF-IoT)*, pages 566–571, 2019. doi: 10.1109/WF-IoT.2019.8767313.

[199] David L. Parnas. Information Distribution Aspects of Design Methodology. Technical report, Departement of Computer Science, Carnegie Mellon University, Pittsburgh, 1971.

[200] Jeffrey Pennington, Richard Socher, and Christopher Manning. GloVe: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar, October 2014. Association for Computational Linguistics. doi: 10.3115/v1/D14-1162. URL https://www.aclweb.org/anthology/D14-1162.

[201] Simone Pettigrew and Stephen Charters. Tasting as a projective technique. *Qualitative Market Research: An International Journal*, 11(3):331–343, 2008.

[202] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. Mining StackOverflow to Turn the IDE into a Self-confident Programming Prompter. In *11th Working Conference on Mining Software Repositories*, pages 102–111, New York, 2014. ACM. ISBN 978-1-4503-2863-0. doi: 10.1145/2597073.2597077.

[203] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. Prompter - turning the IDE into a self-confident programming assistant. *Empirical Software Engineering*, 21(5):2190–2231, 2016. doi: 10.1007/s10664-015-9397-1. URL https://doi.org/10.1007/s10664-015-9397-1.

[204] Luca Ponzanelli, Simone Scalabrino, Gabriele Bavota, Andrea Mocci, Rocco Oliveto, Massimiliano Di Penta, and Michele Lanza. Supporting Software Developers with a Holistic Recommender System. In *39th International Conference on Software Engineering*, pages 94–105, Piscataway, 2017. IEEE. ISBN 978-1-5386-3868-2. doi: 10.1109/ICSE.2017.17.

[205] M.F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, January 1980. ISSN 0033-0337. doi: 10.1108/eb046814. URL https://doi.org/10.1108/eb046814. Publisher: MCB UP Ltd.

[206] Gede Artha Azriadi Prana, Christoph Treude, Ferdian Thung, Thushari Atapattu, and David Lo. Categorizing the content of github README files. *Empirical Software Engineering*, 24(3):1296–1327, 2019. doi: 10.1007/s10664-018-9660-3. URL https://doi.org/10.1007/s10664-018-9660-3.

[207] Sebastian Proksch, Veronika Bauer, and Gail C. Murphy. How to build a recommendation system for software engineering. In Bertrand Meyer and Martin Nordio, editors, *Advances in the theory and practice of software engineering - LASER 2013-2014*, volume 8987 of *LNCS*, pages 1–42. Springer, 2015. URL http://tubiblio.ulb.tu-darmstadt.de/77729/.

[208] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. SWIM: Synthesizing What I Mean: Code Search and Idiomatic Snippet Synthesis. In *38th International Conference on Software Engineering*, pages 357–367, New York, 2016. ACM. ISBN 978-1-4503-3900-1. doi: 10.1145/2884781.2884808.

[209] Mohammad Rahman, Shamima Yeasmin, and Chanchal Roy. Towards a Context-Aware IDE-Based Meta Search Engine for Recommendation about Programming Errors and Exceptions. In *Conference on Software Maintenance, Reengineering, and Reverse Engineering*, pages 194–203, Piscataway, 2014. IEEE. doi: 10.1109/CSMR-WCRE.2014.6747170.

[210] Sebastian Raschka. Model evaluation, model selection, and algorithm selection in machine learning. *CoRR*, abs/1811.12808, 2018. URL http://arxiv.org/abs/1811.12808.

[211] Veselin Raychev, Martin Vechev, and Eran Yahav. Code Completion with Statistical Language Models. In *35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 419–428, New York, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594321.

[212] Jason D. M. Rennie, Lawrence Shih, Jaime Teevan, and David R. Karger. Tackling the poor assumptions of naive bayes text classifiers. In *Proceedings of the Twentieth International Conference on International Conference on Machine Learning*, ICML'03, page 616–623. AAAI Press, 2003. ISBN 1577351894.

[213] Francesco Ricci, Lior Rokach, and Bracha Shapira. *Introduction to Recommender Systems Handbook*, pages 1–35. Springer US, Boston, MA, 2011. ISBN 978-0-387-85820-3. doi: 10.1007/978-0-387-85820-3_1. URL https://doi.org/10.1007/978-0-387-85820-3_1.

[214] C. Richardson and J. R. Rymer. The forrester wave: Low-code development platforms, q2 2016, April 2016. Technical report, Forrester Research.

[215] C. Richardson and J. R. Rymer. Vendor Landscape: The Fractured, Fertile Terrain Of Low-Code Application Platforms. page 23, 2016.

[216] Kaspar Riesen and Horst Bunke. *Graph Classification and Clustering Based on Vector Space Embedding*. World Scientific Publishing Co., Inc., USA, 2010. ISBN 9789814304719.

[217] Peter C. Rigby and Martin P. Robillard. Discovering Essential Code Elements in Informal Documentation. In *35th International Conference on Software Engineering*, pages 832–841, Piscataway, 2013. IEEE. ISBN 978-1-4673-3076-3.

[218] Martin P Robillard. What Makes APIs Hard to Learn? Answers from Developers. *IEEE software*, 26(6):27–34, 2009.

[219] Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. Automated API property inference techniques. *IEEE Transactions on Software Engineering*, 39(5):613–637, May 2013. ISSN 0098-5589.

[220] Martin P. Robillard, Walid Maalej, Robert J. Walker, and Thomas Zimmermann, editors. *Recommendation Systems in Software Engineering*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. ISBN 978-3-642-45134-8 978-3-642-45135-5. doi: 10.1007/978-3-642-45135-5. URL http://link.springer.com/10.1007/978-3-642-45135-5.

[221] Md Omar Faruk Rokon, Risul Islam, Ahmad Darki, Evangelos E. Papalexakis, and M. Faloutsos. Sourcefinder: Finding malware source-code from publicly available repositories. *ArXiv*, abs/2005.14311, 2020.

[222] Israel J. Mojica Ruiz, Meiyappan Nagappan, Bram Adams, and Ahmed E. Hassan. Understanding reuse in the Android Market. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 113–122, Passau, Germany, June 2012. IEEE. ISBN 978-1-4673-1216-5 978-1-4673-1213-4 978-1-4673-1215-8. doi: 10.1109/ICPC.2012.6240477.

[223] Naiyana Sahavechaphan and Kajal Claypool. Xsnippet: Mining for sample code. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, page 413–430, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933484. doi: 10.1145/1167473.1167508.

[224] Naiyana Sahavechaphan and Kajal Claypool. Xsnippet: Mining for sample code. *SIGPLAN Not.*, 41(10):413–430, October 2006. ISSN 0362-1340. doi: 10.1145/1167515.1167508.

[225] Apurvanand Sahay, Arsene Indamutsa, Davide Di Ruscio, and Alfonso Pierantonio. Supporting the understanding and comparison of low-code development platforms. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 171–178, Portoroz, Slovenia, August 2020. IEEE. ISBN 978-1-72819-532-2. doi: 10.1109/SEAA51224.2020.00036.

[226] Apurvanand Sahay, Arsene Indamutsa, Davide Di Ruscio, and Alfonso Pierantonio. Supporting the understanding and comparison of low-code development platforms. In *46th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2020 - to appear*, 2020.

[227] M. A. Saied, O. Benomar, H. Abdeen, and H. Sahraoui. Mining Multi-level API Usage Patterns. In *22nd International Conference on Software Analysis, Evolution, and Reengineering*, pages 23–32, Piscataway, 2015. IEEE. doi: 10.1109/SANER.2015.7081812.

[228] Mohamed Aymen Saied, Hani Abdeen, Omar Benomar, and Houari Sahraoui. Could We Infer Unordered API Usage Patterns Only Using the Library Source Code? In *23rd International Conference on Program Comprehension*, pages 71–81, Piscataway, 2015. IEEE. doi: 10.1109/ICPC.2015.16.

[229] Mohamed Aymen Saied, Ali Ouni, Houari Sahraoui, Raula Gaikovina Kula, Katsuro Inoue, and David Lo. Improving reusability of software libraries through usage pattern mining. *Journal of Systems and Software*, 145:164 – 179, 2018. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2018.08.032. URL http://www.sciencedirect.com/science/article/pii/S0164121218301699.

[230] Rijul Saini, Gunter Mussbacher, Jin L. C. Guo, and Jörg Kienzle. Domobot: A bot for automated and interactive domain modelling. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381352. doi: 10.1145/3417990.3421385. URL https://doi-org.univaq.clas.cineca.it/10.1145/3417990.3421385.

[231] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th International Conference on World Wide Web*, WWW '01, pages 285–295, New York, NY, USA, 2001. ACM. ISBN 1-58113-348-0. doi: 10.1145/371920.372071. URL http://doi.acm.org/10.1145/371920.372071.

[232] Cezar Sas and Andrea Capiluppi. Labelgit: A dataset for software repositories classification using attributed dependency graphs, 03 2021.

[233] Anand Ashok Sawant and Alberto Bacchelli. fine-GRAPE: fine-grained APi usage extractor – an approach and dataset to investigate API usage. *Empirical Software Engineering*, 22(3):1348–1371, June 2017. ISSN 1382-3256, 1573-7616. doi: 10.1007/s10664-016-9444-6. URL http://link.springer.com/10.1007/s10664-016-9444-6.

[234] Giuseppe Scanniello, Simone Romano, Davide Fucci, Burak Turhan, and Natalia Juristo. Students' and professionals' perceptions of test-driven development: A focus group study. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, page 1422–1427, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450337397. doi: 10.1145/2851613.2851778. URL https://doi.org/10.1145/2851613.2851778.

[235] J. Ben Schafer, Dan Frankowski, Jon Herlocker, and Shilad Sen. *Collaborative Filtering Recommender Systems*, pages 291–324. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-72079-9. doi: 10.1007/978-3-540-72079-9_9. URL https://doi.org/10.1007/978-3-540-72079-9_9.

[236] Markus Schedl, Hamed Zamani, Ching-Wei Chen, Yashar Deldjoo, and Mehdi Elahi. Current challenges and visions in music recommender systems research. *Int. J. Multim. Inf. Retr.*, 7(2):95–116, 2018. doi: 10.1007/s13735-018-0154-2. URL https://doi.org/10.1007/s13735-018-0154-2.

[237] G. L. Scoccia, S. Ruberto, I. Malavolta, M. Autili, and P. Inverardi. An investigation into android run-time permissions from the end users' perspective. In *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 45–55, 2018.

[238] Sagar Sen, Benoit Baudry, and Hans Vangheluwe. Towards domain-specific model editors with automatic model completion. *SIMULATION*, 86(2):109–126, 2010. doi: 10.1177/0037549709340530. URL https://doi.org/10.1177/0037549709340530.

[239] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-lehman graph kernels. *J. Mach. Learn. Res.*, 12 (null):2539–2561, November 2011. ISSN 1532-4435.

[240] David Shriver, Sebastian Elbaum, and Matthew B. Dwyer. Reducing DNN Properties to Enable Falsification with Adversarial Attacks. In *43rd International Conference on Software Engineering*. IEEE, 2021.

[241] Mingdan Si and Qingshan Li. Shilling attacks against collaborative recommender systems: a review. *Artif. Intell. Rev.*, 53(1):291–319, 2020. doi: 10.1007/s10462-018-9655-x. URL https://doi.org/10.1007/s10462-018-9655-x.

[242] Giannis Siglidis, Giannis Nikolentzos, Stratis Limnios, Christos Giatsidis, Konstantinos Skianis, and Michalis Vazirgiannis. GraKeL: A Graph Kernel Library in Python. *arXiv:1806.02193 [cs, stat]*, March 2020. URL http://arxiv.org/abs/1806.02193. arXiv: 1806.02193.

[243] Jagsir Singh and Jaswinder Singh. Detection of malicious software by analyzing the behavioral artifacts using machine learning algorithms. *Information and Software Technology*, 121:106273, May 2020. ISSN 0950-5849. doi: 10.1016/j.infsof.2020.106273. URL https://www.sciencedirect.com/science/article/pii/S0950584920300239.

[244] D.I.K. Sjoeberg, J.E. Hannay, O. Hansen, V.B. Kampenes, A. Karahasanovic, N.-K. Liborg, and A.C. Rekdal. A survey of controlled experiments in software engineering. *IEEE Transactions on Software Engineering*, 31(9):733–753, 2005. doi: 10.1109/TSE.2005.97.

[245] Marcus Soll and Malte Vosgerau. Classifyhub: An algorithm to classify github repositories. pages 373–379, 09 2017. ISBN 978-3-319-67189-5. doi: 10.1007/978-3-319-67190-1_34.

[246] Qinbao Song, Xiaoyan Zhu, Guangtao Wang, Heli Sun, He Jiang, Chenhao Xue, Baowen Xu, and Wei Song. A machine learning based software process model recommendation method. *Journal of Systems and Software*, 118:85–100, 2016.

[247] Charles Spearman. The proof and measurement of association between two things. *The American journal of psychology*, 15(1):72–101, 1904.

[248] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[249] M. Stephan. Towards a Cognizant Virtual Software Modeling Assistant using Model Clones. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 21–24, May 2019. doi: 10.1109/ICSE-NIER.2019.00014.

[250] V. Subramaniyaswamy and R. Logesh. Adaptive KNN based Recommender System through Mining of User Preferences. *Wireless Personal Communications*, 97(2):2229–2247, November 2017. ISSN 0929-6212, 1572-834X. doi: 10.1007/s11277-017-4605-5. URL http://link.springer.com/10.1007/s11277-017-4605-5.

[251] Mahito Sugiyama and Karsten M. Borgwardt. Halting in random walk kernels. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, page 1639–1647, Cambridge, MA, USA, 2015. MIT Press.

[252] Arvind Sujeeth, HyoukJoong Lee, Kevin Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand Atreya, Martin Odersky, and Kunle Olukotun. OptiML: an implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 609–616, 2011.

[253] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, page 3104–3112, Cambridge, MA, USA, 2014. MIT Press.

[254] Watanabe Takuya and Hidehiko Masuhara. A Spontaneous Code Recommendation Tool Based on Associative Search. In *3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation*, pages 17–20, New York, 2011. ACM. ISBN 978-1-4503-0597-6. doi: 10.1145/1985429.1985434.

[255] Thomas Thum, Christian Kastner, Sebastian Erdweg, and Norbert Siegmund. Abstract features in feature modeling. In *2011 15th International Software Product Line Conference*, pages 191–200, 2011. doi: 10.1109/SPLC.2011.53.

[256] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. Featureide: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79:70–85, 2014.

[257] S. Thummalapenta and Tao Xie. SpotWeb: Detecting Framework Hotspots and Coldspots via Mining Open Source Code on the Web. In *23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 327–336, Washington, 2008. IEEE. ISBN 978-1-4244-2187-9.

[258] Suresh Thummalapenta, Luigi Cerulo, Lerina Aversano, and Massimiliano Di Penta. An empirical study on the maintenance of source code clones. *Empirical Softw. Engg.*, 15(1):1–34, February 2010. ISSN 1382-3256. doi: 10.1007/s10664-009-9108-x.

[259] Ferdian Thung, David Lo, and Julia Lawall. Automated library recommendation. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 182–191, Oct 2013. doi: 10.1109/WCRE.2013.6671293.

[260] Liang Tong, Bo Li, Chen Hajaj, Chaowei Xiao, Ning Zhang, and Yevgeniy Vorobeychik. Improving robustness of ml classifiers against realizable evasion attacks using conserved features. In *USENIX Security Symposium*, pages 285–302, 2019.

[261] Christoph Treude and Martin P. Robillard. Augmenting API Documentation with Insights from Stack Overflow. In *38th International Conference on Software Engineering*, pages 392–403, New York, 2016. ACM. ISBN 978-1-4503-3900-1. doi: 10.1145/2884781.2884800.

[262] J. D. Tygar. Adversarial machine learning. *IEEE Internet Computing*, 15(5):4–6, 2011.

[263] Gias Uddin and Martin P Robillard. How API Documentation Fails. *IEEE Software*, 32(4):68–75, 2015. ISSN 0740-7459. doi: 10.1109/MS.2014.80.

[264] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in science & engineering*, 13(2):22–30, 2011.

[265] Saúl Vargas and Pablo Castells. Rank and relevance in novelty and diversity metrics for recommender systems. In *Proceedings of the Fifth ACM Conference on Recommender Systems*, RecSys '11, page 109–116, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450306836. doi: 10.1145/2043932.2043955. URL https://doi.org/10.1145/2043932.2043955.

[266] Saúl Vargas and Pablo Castells. Improving sales diversity by recommending users to items. In *Proceedings of the 8th ACM Conference on Recommender Systems*, RecSys '14, page 145–152, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450326681. doi: 10.1145/2645710.2645744.

[267] Saúl Vargas. Novelty and diversity enhancement and evaluation in recommender systems and information retrieval. In *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*, SIGIR '14, page 1281, New York, NY, USA, July 2014. Association for Computing Machinery. ISBN 978-1-4503-2257-7. doi: 10.1145/2600428.2610382. URL https://doi.org/10.1145/2600428.2610382.

[268] Saúl Vargas and Pablo Castells. Rank and relevance in novelty and diversity metrics for recommender systems. In *Proceedings of the fifth ACM conference on recommender systems*, RecSys '11, pages 109–116, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0683-6.

[269] Saúl Vargas and Pablo Castells. Improving sales diversity by recommending users to items. In *Eighth ACM conference on recommender systems, RecSys '14, foster city, silicon valley, CA, USA - october 06 - 10, 2014*, pages 145–152, 2014.

[270] S. Vargas-Baldrich, M. Linares-Vásquez, and D. Poshyvanyk. Automated Tagging of Software Projects Using Bytecode and Dependencies. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 289–294, November 2015. doi: 10.1109/ASE.2015.38.

[271] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[272] C. Velázquez-Rodríguez and C. De Roover. MUTAMA: An Automated Multi-label Tagging Approach for Software Libraries on Maven. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 254–258, September 2020. doi: 10.1109/SCAM51674.2020.00034. ISSN: 2470-6892.

[273] Nicolas Viennot, Edward Garcia, and Jason Nieh. A measurement study of google play. In *The 2014 ACM international conference on Measurement and modeling of*

computer systems - SIGMETRICS '14, pages 221–233, Austin, Texas, USA, 2014. ACM Press. ISBN 978-1-4503-2789-3.

[274] S.V.N. Vishwanathan, Nicol N. Schraudolph, Risi Kondor, and Karsten M. Borgwardt. Graph kernels. *Journal of Machine Learning Research*, 11(40):1201–1242, 2010. URL http://jmlr.org/papers/v11/vishwanathan10a.html.

[275] J. Wang and J. Han. Bide: efficient mining of frequent closed sequences. In *Proceedings. 20th International Conference on Data Engineering*, pages 79–90, 2004. doi: 10.1109/ICDE.2004.1319986.

[276] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang. Mining Succinct and High-coverage API Usage Patterns from Source Code. In *10th MSR*, pages 319–328, Piscataway, 2013. IEEE. doi: 10.1109/MSR.2013.6624045.

[277] Jianfang Wang and Pengfei Han. Adversarial Training-Based Mean Bayesian Personalized Ranking for Recommender System. *IEEE Access*, 8:7958–7968, 2020. ISSN 2169-3536. doi: 10.1109/ACCESS.2019.2963316. Conference Name: IEEE Access.

[278] Kaiyuan Wang, Allison Sullivan, D. Marinov, and S. Khurshid. Asketch: a sketching framework for alloy. *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018.

[279] Tao Wang, Huaimin Wang, Gang Yin, Charles X. Ling, Xiao Li, and Peng Zou. Tag recommendation for open source software. *Frontiers of Computer Science*, 8(1):69–82, February 2014. ISSN 2095-2228, 2095-2236. doi: 10.1007/s11704-013-2394-x. URL http://link.springer.com/10.1007/s11704-013-2394-x.

[280] Ting-Hsiang Wang, Xia Hu, Haifeng Jin, Qingquan Song, Xiaotian Han, and Zirui Liu. Autorec: An automated recommender system. In *Fourteenth ACM Conference on Recommender Systems*, RecSys '20, page 582–584, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375832. doi: 10.1145/3383313.3411529.

[281] Robert Waszkowski. Low-code platform for automating business processes in manufacturing. *IFAC-PapersOnLine*, 52(10):376–381, 2019. ISSN 24058963. doi: 10.1016/j.ifacol.2019.10.060.

[282] Boris Weisfeiler and Andrei Leman. The reduction of a graph to canonical form and the algebra which appears therein. *NTI, Series*, 2(9):12–16, 1968.

[283] Karl Weiss, Taghi Khoshgoftaar, and DingDing Wang. A survey of transfer learning. *Journal of Big Data*, 3, 12 2016. doi: 10.1186/s40537-016-0043-6.

[284] Martin Weyssow, Houari A. Sahraoui, and Eugene Syriani. Recommending metamodel concepts during modeling activities with pre-trained language models. *Softw. Syst. Model.*, 21(3):1071–1089, 2022. doi: 10.1007/s10270-022-00975-5. URL https://doi.org/10.1007/s10270-022-00975-5.

[285] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1 (6):80–83, 1945. ISSN 00994987.

[286] D.H. Wolpert and W.G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, April 1997. ISSN 1089778X. doi: 10.1109/4235.585893. URL http://ieeexplore.ieee.org/document/585893/.

[287] Tzu-Tsung Wong. Performance Evaluation of Classification Algorithms by K-fold and Leave-one-out Cross Validation. *Pattern Recognition*, 48(9):2839–2846, 2015. ISSN 0031-3203. doi: 10.1016/j.patcog.2015.03.009.

[288] Daoyuan Wu, Debin Gao, Rocky K. C. Chang, En He, Eric K. T. Cheng, and Robert H. Deng. Understanding open ports in android applications: Discovery, diagnosis, and security assessment. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019. URL https://bit.ly/3e3enkJ.

[289] Songyang Wu, Pan Wang, Xun Li, and Yong Zhang. Effective detection of android malware based on the usage of data flow APIs and machine learning. *Information and Software Technology*, 75:17–25, July 2016. ISSN 09505849. doi: 10.1016/j.infsof.2016.03.004. URL https://linkinghub.elsevier.com/retrieve/pii/S0950584916300386.

[290] Frank F. Xu, Bogdan Vasilescu, and Graham Neubig. In-ide code generation from natural language: Promise and challenges. *CoRR*, abs/2101.11149, 2021. URL https://arxiv.org/abs/2101.11149.

[291] Qiuling Xu, Guanhong Tao, Siyuan Cheng, and Xiangyu Zhang. Towards feature space adversarial attack by style perturbation. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(12):10523–10531, May 2021. doi: 10.1609/aaai.v35i12.17259. URL https://ojs.aaai.org/index.php/AAAI/article/view/17259.

[292] He Zhang, Muhammad Ali Babar, and Paolo Tell. Identifying relevant studies in software engineering. *Information and Software Technology*, 53(6):625–637, 2011.

[293] Yiming Zhang, Yujie Fan, Shifu Hou, Yanfang Ye, Xusheng Xiao, Pan Li, Chuan Shi, Liang Zhao, and Shouhuai Xu. Cyber-guided Deep Neural Network for Malicious Repository Detection in GitHub. In *2020 IEEE International Conference on Knowledge Graph (ICKG)*, pages 458–465, August 2020. doi: 10.1109/ICBK50248.2020.00071.

[294] Yu Zhang, Frank Xu, Sha Li, Yu Meng, Xuan Wang, Qi Li, and Jiawei Han. Higitclass: Keyword-driven hierarchical classification of github repositories, 10 2019.

[295] Yun Zhang, David Lo, Pavneet Singh Kochhar, Xin Xia, Quanlai Li, and Jianling Sun. Detecting similar repositories on GitHub. *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 00:13–23, 2017.

[296] Zhi-Dan Zhao and Ming-sheng Shang. User-based collaborative-filtering recommendation algorithms on hadoop. In *Proceedings of the 2010 Third International Conference on Knowledge Discovery and Data Mining*, WKDD '10, pages 478–481, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-3923-2. doi: 10.1109/WKDD.2010.54. URL https://doi.org/10.1109/WKDD.2010.54.

[297] Mengya Zheng, Xingyu Pan, and David Lillis. CodEX: Source code plagiarism detection based on abstract syntax tree. In *Proceedings for the 26th AIAI irish conference on artificial intelligence and cognitive science trinity college dublin, dublin, ireland, december 6-7th, 2018.*, pages 362–373, 2018.

[298] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. MAPO: Mining and Recommending API Usage Patterns. In *23rd European Conference on Object-Oriented Programming*, pages 318–343, Berlin, Heidelberg, 2009. Springer. ISBN 978-3-642-03012-3. doi: 10.1007/978-3-642-03013-0_15.

[299] Yuqi Zhou, Jiawei Wu, and Yanchun Sun. Ghtrec: A personalized service to recommend github trending repositories for developers. In *2021 IEEE International Conference on Web Services (ICWS)*, pages 314–323, 2021. doi: 10.1109/ICWS53863.2021.00049.

[300] Önder Babur. A labeled Ecore metamodel dataset for domain clustering, March 2019. URL https://doi.org/10.5281/zenodo.2585456.